HZ BOOKS
华章教育

计 算 机 科 学 丛 书

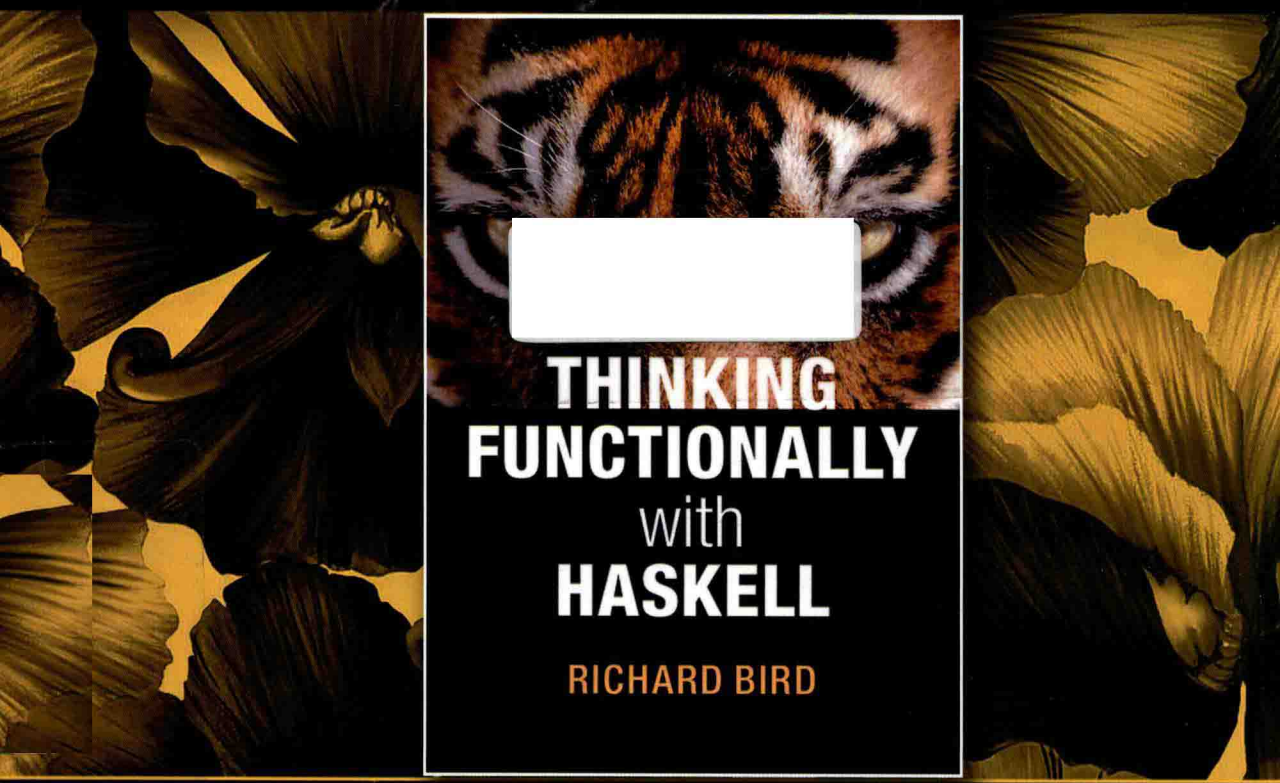
CAMBRIDGE

Haskell函数式程序设计

[英] 理查德·伯德（Richard Bird） 著
牛津大学

乔海燕 译
中山大学

Thinking Functionally with Haskell



机械工业出版社
China Machine Press

Haskell函数式程序设计

Thinking Functionally with Haskell

Richard Bird教授的文字以清晰和严谨著称，他为初学函数式程序设计的学生所著的这本新教材，强调利用数学思维进行推理的基本方法。在解决问题时，首先从显而易见的简单方法入手，然后应用一些熟知的恒等式，运用等式规则逐步推理，最终得到效率倍增的解。在这一过程中，学生不仅理解了程序的性质，而且实现了更高效的计算。

本书特色

- 涵盖Haskell的大量特性，但不拘泥于语言细节，而是借助它来阐明函数式程序设计的思想和方法。
- 包含数独实例和精美打印实例，以及100余道精心挑选的习题，并配有详尽的解答。
- 免费下载源代码：www.cs.ox.ac.uk/publications/books/functional。

作者简介

理查德·伯德（Richard Bird） 牛津大学计算机实验室的荣誉退休教授，牛津大学林肯学院的研究员。他的著述颇丰，包括《Algebra of Programming》(Prentice Hall, 1996)和《Pearls of Functional Algorithm Design》(Cambridge University Press, 2010)。

CAMBRIDGE
UNIVERSITY PRESS
www.cambridge.org

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机\程序设计\函数式程序设计

ISBN 978-7-111-52932-3

9 787111 529323 >

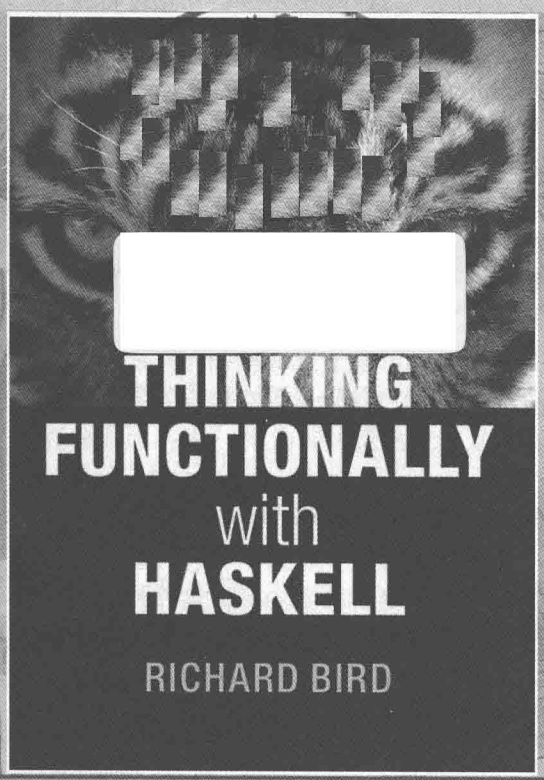
定价：69.00元

计 算 机 科 学 丛 书

Haskell函数式程序设计

[英] 理查德·伯德（Richard Bird） 著 乔海燕 译
牛津大学 中山大学

Thinking Functionally with Haskell



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Haskell 函数式程序设计 /（英）伯德（Bird，R.）著；乔海燕译 . —北京：机械工业出版社，2016.3
（计算机科学丛书）
书名原文：Thinking Functionally with Haskell

ISBN 978-7-111-52932-3

I. H… II. ①伯… ②乔… III. 函数－程序设计 IV. TP311.1

中国版本图书馆 CIP 数据核字（2016）第 028759 号

本书版权登记号：图字：01-2015-5205

本书通过 Haskell 语言介绍函数式程序设计的基本思想和方法，讲解如何将数学思维应用于程序设计问题，以实现更高效的计算。本书涵盖 Haskell 的诸多特性，但并不是这门语言的参考指南，而是旨在利用丰富的实例和练习揭示函数式程序设计的本质。

本书不要求读者具备程序设计基础，所涉及的数学知识也不高深，既适合初学者阅读，也适合有经验的程序员参考。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：曲　熠　　责任校对：董纪丽
印　　刷：中国电影出版社印刷厂　　版　　次：2016 年 3 月第 1 版第 1 次印刷
开　　本：185mm×260mm　1/16　　印　　张：15.25
书　　号：ISBN 978-7-111-52932-3　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

本书是《Introduction to Functional Programming Using Haskell, Second Edition》的全新升级，主要变化有：重新组织部分介绍性内容，以适应一个学期或者两个学期课程的不同需要；几个新的实例；100 多道习题及其答案。与以前的版本一样，本书不需要读者具有计算机或者程序设计知识，因此本书适用于计算专业的第一门课程。

在编写教材时，每个作者各具风格，本书也不例外。尽管现在有很多关于 Haskell 的书、教程、文章和博客等，但是很少有人强调用数学思维思考函数式程序设计的能力，在我看来，正是这种能力使其成为有史以来最棒的程序设计方法。这其中所涉及的数学知识既不新也不复杂，任何学过高中数学（如三角函数）并且应用三角函数恒等式化简过正余弦表达式（一个典型的例子：将 $\sin 3\alpha$ 用 $\sin\alpha$ 来表示）的学生很快会发现，在程序设计问题中所要做的工作是完全类似的。使用函数式程序设计所获得的回报是更快的计算。即使在 30 年后，我依然使用这样的方法，并能从中得到很大的快乐：在解决问题时首先从一个简单、明显却不太高效的方法入手，然后应用一些熟知的恒等式，最后得到一个高效 10 倍的解。当然，如果我运气好的话。

如果上一段的最后一句让你失去兴趣，如果你一直在远离数学的“魔多”（Mordor），那么本书可能不适合你。我只是说有这种可能，但也不一定（没有人愿意失去读者）。我们在学习一种新的、令人兴奋的编程方法时仍能得到不少乐趣。即使是那些因为各种原因在日常工作中不能使用 Haskell，而且也没有时间计算更优解的程序员，仍然因学习 Haskell 所带来的享受而倍受鼓舞，而且非常赞赏它既简单又清晰简洁地表达计算思想和方法的能力。事实上，用纯函数式表达程序设计思想的能力已经慢慢地融入了主流的命令式程序设计语言，如 Python、Visual Basic 和 C#。

最后，也是最重要的一点：Haskell 是一种大规模语言，本书不能涵盖一切内容。本书不是 Haskell 的参考指南。尽管 Haskell 语言的细节在每一页出现，特别是在前几章，但是我的初衷是讲解函数式程序设计的本质，用函数思考程序的思想，而不是赘述一种特定语言的特点。但是，过去几年来 Haskell 已经吸收并实现了早期函数语言（如 SASL、KRC、Miranda、Orwell 和 Gofer）中表达的函数式程序设计的大多数思想，而且难以抵挡用这种超酷语言介绍所有这些特性的诱惑。

书中出现的大多数程序可以在下列网页上找到：

www. cs. ox. ac. uk/publications/books/functional

希望将来有更多习题（及答案）和编程项目的建议等可以添加进来。关于 Haskell 的更多信息，读者应该首选官网 www. haskell. org。

致谢

本书源于我基于第 2 版所写的讲义。来自助教和学生的意见和建议为本书增添了很多

光彩。另有很多读者通过电子邮件给出建设性的评论和批评，或者指出书中的打字错误和低级错误。这些读者包括：Nils Andersen，Ani Calinescu，Franklin Chen，Sharon Curtis，Martin Filby，Simon Finn，Jeroen Fokker，Maarten Fokkinga，Jeremy Gibbons，Robert Giegerich，Kevin Hammond，Ralf Hinze，Gerard Huet，Michael Hinchey，Tony Hoare，Iain Houston，John Hughes，Graham Hutton，Cezar Ionescu，Stephen Jarvis，Geraint Jones，Mark Jones，John Launchbury，Paul Licameli，David Lester，Iain MacCullum，Ursula Martin，Lambert Meertens，Erik Meijer，Quentin Miller，Oege de Moor，Chris Okasaki，Oskar Permvall，Simon Peyton Jones，Mark Ramaer，Hamilton Richards，Dan Russell，Don Sannella，Antony Simmons，Deepak D'Souza，John Spanondakis，Mike Spivey，Joe Stoy，Bernard Sufrin，Masato Takeichi，Peter Thiemann，David Turner，Colin Watson 和 Stephen Wilson。特别是 Jeremy Gibbons、Bernard Sufrin 和 José Pedro Magalhães 阅读了初稿，并提出了许多建议。

感谢剑桥大学出版社编辑 David Tranah 持续不断的建议和支持。我现在是牛津大学计算机系荣誉退休教授，感谢计算机系和系主任 Bill Roscoe 的一贯支持。

格式说明

习题

习题 A　请用 $\sin\alpha$ 表示 $\sin 3\alpha$。

答案

习题 A 答案

$$
\begin{aligned}
& \sin 3\alpha \\
= & \{\text{算术}\} \\
& \sin(2\alpha + \alpha) \\
= & \{\text{因为 } \sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta\} \\
& \sin 2\alpha\cos\alpha + \cos 2\alpha\sin\alpha \\
= & \{\text{因为 } \sin 2\alpha = 2\sin\alpha\cos\alpha\} \\
& 2\sin\alpha\cos^2\alpha + \cos 2\alpha\sin\alpha \\
= & \{\text{因为 } \cos 2\alpha = \cos^2\alpha - \sin^2\alpha\} \\
& 2\sin\alpha\cos^2\alpha + (\cos^2\alpha - \sin^2\alpha)\sin\alpha \\
= & \{\text{因为 } \cos^2\alpha + \sin^2\alpha = 1\} \\
& \sin\alpha(3 - 4\sin^2\alpha)
\end{aligned}
$$

以上证明格式是由 Wim Feijen 发明的，本书将使用这种证明格式。

Richard Bird

第1章
Thinking Functionally with Haskell

何谓函数式程序设计

简而言之：

- 函数式程序设计是一种构造程序的方法，它强调的是函数和函数的应用，而非命令及其运行。
- 函数式程序设计使用简单的数学语言，使得问题的描述更清晰也更简洁。
- 函数式程序设计的数学基础简单，而且支持对函数的性质进行推理。

本书的目的是使用一种称为 Haskell 的函数语言展示以上 3 个特性。

1.1 函数和类型

本书将使用 Haskell 的如下记法：

```
f :: X -> Y
```

它表示 f 是一个函数，其参数类型是 X，返回值的类型是 Y。例如：

```
sin     :: Float -> Float
age     :: Person -> Int
add     :: (Integer,Integer) -> Integer
logBase :: Float -> (Float -> Float)
```

Float 表示像 3.14159 等浮点数类型；Int 表示有限精度整数类型，即满足 $-2^{29} \leqslant n < 2^{29}$ 的整数 n；Integer 表示无精度限制整数类型。在第 3 章将会看到，Haskell 包含了各种数值类型。

数学上用 $f(x)$ 表示将函数 f 应用于其参数 x。但是，也使用如 $\sin\theta$ 来表示 $\sin(\theta)$。在 Haskell 中可以始终使用 f x 表示将 f 应用于参数 x。函数的应用运算用一个空格表示。如果不使用括号，那么必须用空格避免多字母名可能引起的混淆：latex 是一个名，但是 late x 表示函数 late 应用于参数 x。

例如，sin 3.14、sin (3.14)或 sin(3.14)是函数 sin 应用于 3.14 的 3 种合法表示。

类似地，logBase 2 10、(logBase 2) 10 或(logBase 2)(10)都是以 2 为底 10 的对数的正确表示。但是，表达式 logBase (2 10)是错误的。式子 add (3,4)表示 3 与 4 之和，其中的括号是必需的，因为 add 的参数类型是一对整数，而且数对需要用括号和逗号表示。

再看看 logBase 的类型，其参数是一个浮点数，返回值是一个函数。初看起来可能有些奇怪，但再细看则不然：这里的 logBase 2 和 logBase e 恰好表示了数学函数 $\log_2$ 和 $\log_e$。

数学上有形如 log sin x 的表达式。对于数学家来讲，该式子表示 log (sin x)，因为(log sin) x 没有意义。但是在 Haskell 中，必须说明一个式子的含义，而且必须将

该式子写成 `log (sin x)`，因为 Haskell 将 `log sin x` 解释为 `(log sin) x`。在 Haskell 表达式中函数应用是左结合的，而且具有最高的优先级。（此外，`log` 是 `logBase e` 在 Haskell 中的简写。）

下面是另一个例子。在三角函数中，$\sin 2\theta = 2\sin\theta\cos\theta$。在 Haskell 中该式子写成

```
sin (2*theta) = 2 * sin theta * cos theta
```

我们不仅要显式地表示乘法，而且要使用括号表达确切含义。上式也可以添加更多括号，写成

2

```
sin (2*theta) = 2 * (sin theta) * (cos theta)
```

但是，多加的括号不是必需的，因为函数应用的优先级比乘法的优先级高。

1.2 函数复合

假设 `f :: Y -> Z` 和 `g :: X -> Y` 是两个函数，可以将这两个函数复合成一个新的函数：

```
f . g :: X -> Z
```

该函数将 `g` 应用于类型 `X` 的参数，得到类型 `Y` 的结果，然后将 `f` 应用于这个结果，最后得到类型 `Z` 的结果。我们将始终使用这样的术语：函数输入参数，返回结果。事实上，有

```
(f . g) x = f (g x)
```

复合的顺序是从右到左，这是因为我们把函数写在其应用的参数左边。英语的“green pig”中形容词“green”解释为函数，它应用于名词短语，得到名词短语。当然，在法语中情况相反。

1.3 例子：高频词

下面通过解决一个问题来说明函数复合的重要性。《战争与和平》中出现最多的 100 个词是哪些？《爱的徒劳》中出现最多的 50 个词是什么？下面将设计一个函数程序求得答案。不过，尽管现在还没到编写一个完整程序的时候，但是，可以通过构造足够的成分来展示函数式程序设计的精髓。

给定的输入是什么？答：一个文本，可视作由字符构成的一个列表。这里的字符既包含可见字符如`'B'`和`','`，也包含空白字符（blank character），如空格和换行符（`' '`和`'\n'`）。注意，单个字符用单引号表示。例如，`'f'`是一个字符，而 `f` 是一个名。Haskell 用 `Char` 表示字符类型，元素类型为 `Char` 的列表类型用`[Char]`表示。这种记法不只适用于字符，例如，`[Int]`表示整数列表，`[Float -> Float]`表示函数列表。

期待的输出是什么？答：如下形式的数据。

3

```
the: 154
of: 50
a: 18
and: 12
in: 11
```

以上显示也是一个字符列表，事实上可看成如下列表：

```
"  the: 154\n  of: 50\n  a: 18\n  and: 12\n  in: 11\n"
```

字符的列表用双引号表示。更多列表知识参见习题。所以，我们需要设计一个函数，不妨称为 commonWords，其类型为

```
commonWords :: Int -> [Char] -> [Char]
```

函数 commonWords n 获得一个字符列表作为输入，返回该列表中 n 个出现最多的词构成的串（字符列表的别名），形如前面所述列表。commonWords 的类型没有使用括号，当然也可以写成

```
commonWords :: Int -> ([Char] -> [Char])
```

当一个类型中有两个相邻的符号 ->时，结合的顺序是自右向左，与函数应用的结合顺序恰恰相反。因此，A -> B -> C 表示A -> (B -> C)。如果想表示类型(A -> B) -> C，那么必须使用括号。更多相关知识参见第 2 章。

明白了给定的输入和期待的输出后，不同的人有不同的解法，对问题的关注点也不尽相同。例如，什么是一个“词”？如何将字符列表转换为词的列表？"Hello"、"hello"以及"Hello!"是不同的词还是相同的词？如何计算词的数目？需要统计所有的词数，还是只要计算最常出现的词数？等等。有些人觉得这些过多的细节使人望而却步，大多数人似乎认为在计算过程中，某个时刻必须获得词与其出现频率的列表，但是如何由该列表实现最终目标呢？是扫描该列表 n 次，每次找出下一个出现次数最多的词，还是有其他更好的方法？

首先考虑词的概念，并简单地假定一个词是不含空格和换行符的最大字符序列。这样的定义允许把诸如"Hello!"、"3*4"和"Thelma&Louise"等看作词，但是这没关系。在一个文本中，一个词是有空白字符包围的字符序列，如"Thelma and Louise"包含 3 个词。

我们不准备考虑如何将一个文本分解成其组成元素（即词的列表），而是假定存在具有这种功能的函数：

```
words :: [Char] -> [[Char]]
```

诸如[[Char]]这样的类型显得难以记忆，不过在 Haskell 中总是可以引入类型同义词（type synonyms）：

```
type Text = [Char]
type Word = [Char]
```

现在可以这样表达类型 words :: Text -> [Word]，使其更便于记忆。当然，一个文本有别于一个词，前者可以包含空白字符，后者则不然，但是 Haskell 的类型同义词不能表达这种细微的区别。事实上，words 是 Haskell 的库函数，因此不必自定义。

另外一个问题是"The"和"the"是否表示同一个词。它们实际上是同一个词，解决这个问题的一种方法是将文本中的所有字母都转换成小写，其他字符不变。为此，需要一个函数 toLower :: Char -> Char，该函数将大写字母转换成小写字母，其他字符保持不变。为了将该函数应用于文本的每个字符，需要下面的通用函数：

```
map :: (a -> b) -> [a] -> [b]
```

使得 map f 应用于一个列表时，f 被应用于列表的每个元素。这样，将每个字母转换为小写由下列函数完成：

```
map toLower :: Text -> Text
```

好了，现在得到了将文本转换为小写字母词的列表函数 words . map toLower。下一个任务是计算每个词出现的次数。可以扫描词的列表，检查下一个词是第一次出现还是已经出现过，相应地开始新词的计数或者给对应词的计数器加 1。不过，另一种更简单的想法是对词的列表按照字典序排序，结果是所有重复出现的词排在了一起。人工操作时不会这样做，但是通过排序获得信息的思想或许是计算过程中最重要的算法思想。所以，假设存在一个函数：

```
sortWords :: [Word] -> [Word]
```

5 该函数将词的列表按照字典序排序。例如：

```
sortWords ["to","be","or","not","to","be"]
       = ["be","be","not","or","to","to"]
```

下一步需要计算在有序列表中每个词连续出现的次数。假定已有计算词数的函数：

```
countRuns :: [Word] -> [(Int,Word)]
```

例如：

```
countRuns ["be","be","not","or","to","to"]
  = [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
```

其结果是按字典序排列的词及其出现次数的列表。

现在来考虑关键的思想：希望数据按照词的出现次数从大到小排列，而不是按照词的字典序排列。可以看出，这就是一种排序，无需设计其他更聪明的方法。如前所述，排序确实是程序设计中非常有用的方法。因此，假定已有函数：

```
sortRuns :: [(Int,Word)] -> [(Int,Word)]
```

该函数将词及其出现次数按照出现次数（列表元素的第一个分量）递减排序。例如：

```
sortRuns [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
      = [(2,"be"),(2,"to"),(1,"not"),(1,"or")]
```

接下来只需取出结果列表中的前 n 个元素。为此，需要下列函数：

```
take :: Int -> [a] -> [a]
```

该函数使得 take n 取得一个列表的前 n 个元素。函数 take 并不关心列表中的元素是什么类型，这就是 take 的类型签名中出现了 a，而不是(Int,Word)的原因。第 2 章将解释这种思想。

最后的步骤仅仅是整理格式。首先将每个元素转换成一个串，例如，将(2,"be")转换为"be 2 \n"。将该函数称为

```
showRun :: (Int,Word) -> String
```

类型 String 是 Haskell 的预定义类型，实际上是[Char]的类型同义词。因此，下列函数将词及其次数列表转换为串列表：

```
map showRun :: [(Int,Word)] -> [String]
```

最后一步需要使用下列函数：

6

```
concat :: [[a]] -> [a]
```

该函数将元素的列表的列表串联成一个列表。同样，函数 concat 并不关心这里串联的是什么“元素”，这也是类型中出现 a 的原因。

下面定义函数：

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

函数 commonWords 定义中使用了 8 个分函数，并用函数复合将它们管道式粘合起来。并非每个问题都可以这样直接地分解成一系列子问题，但是，如果可行的话，最后的程序将是简单、迷人而且有效的。

需要注意的是分解问题的过程是如何在辅助函数的类型指导下进行的。第二个经验（第一个经验是函数复合的重要性）是，确定一个函数的类型是找到该函数合适定义的第一步。

本节的目的是设计一个解决高频词问题的程序。结果是利用辅助函数给出了 commonWords 的函数定义，这些辅助函数或者可以直接定义，或者由某个 Haskell 函数库提供。脚本（script）是一些定义的集合，所以，我们实际上构造了一个脚本。脚本中函数定义的顺序并不重要。函数 commonWords 的定义完全可以放在最前面，然后再定义辅助函数，或者先定义辅助函数，最后给出主要函数的定义。换言之，程序员可以在脚本中用任何顺序叙述故事。稍后将解释如何用脚本进行计算。

1.4 例子：数字转换为词

本节讨论另一个例子，并给出完整解。这个例子展示了求解问题的另一个基本方面，即解决一个复杂问题的好方法，首先是简化问题，然后考虑如何解决更简单的问题。

有时需要把数字写成词。例如：

7

What is functional programming?

```
convert 308000 = "three hundred and eight thousand"
convert 369027 = "three hundred and sixty-nine thousand and
                  twenty-seven"
convert 369401 = "three hundred and sixty-nine thousand
                  four hundred and one"
```

我们的目标是设计这样一个函数：

```
convert :: Int -> String
```

即对于一个给定的不超过 100 万的非负数，函数返回用词表示的数字。如上所述，String 是 Haskell 预定义的类型[Char]的同义词。

这里需要其中各个数字的名称。一种方法是用串的列表表示它们：

```
> units, teens, tens :: [String]
> units = ["zero","one","two","three","four","five",
```

```
>          "six","seven","eight","nine"]
> teens = ["ten","eleven","twelve","thirteen","fourteen",
>          "fifteen","sixteen","seventeen","eighteen",
>          "nineteen"]
> tens  = ["twenty","thirty","forty","fifty","sixty",
>     .    "seventy","eighty","ninety"]
```

以上每行开始的字符 > 表示什么？答案是，在一个脚本中，该字符表示一行 Haskell 代码，而不是注释。用 .lhs 做扩展名的 Haskell 文件称为 Haskell 文学脚本（Literate Haskell Script），习惯上脚本的每一行都是注释，除非出现符号 >，该符号表示随后的是 Haskell 代码行。Haskell 不允许代码行和注释紧邻，所以代码行和注释之间至少应该有一行空白。事实上，你正在阅读的本章就是一个合法的 .lhs 文件，完全可以将该文件载入 Haskell 系统并交互运行。在今后的章节将不再延续这种传统（除此之外，我们被迫用不同的名表示一个函数的不同定义），但是，本章展示的文学编程允许使用任何顺序讨论和书写函数的定义。

对于当前的任务，解决复杂问题的一个好方法是先解决一个更简单的问题。该问题的最简单情况是给定的数字只有一位数，即 $0 \leqslant n < 10$。假定用 convert1 解决这种简单情
8 况。现在马上可以定义：

```
> convert1 :: Int -> String
> convert1 n = units!!n
```

这个定义使用了列表索引运算（!!）。对于给定的列表 xs 和下标 n，表达式 xs!!n 返回 xs 中位置为 n 的元素，其中位置从 0 开始计算。特别地，units!!0 = "zero"。而且，units!!10 确实无定义，因为 units 只有 10 个元素，下标是 0 ~ 9。一般地，在一个脚本中定义的函数是*部分函数*或*不完全函数*，即并非对每个参数返回确切定义的结果。

这个问题的下一个最简单情况是数字 n 最多有两位数，即 $0 \leqslant n < 100$。假定 convert2用于处理这种情况。因为需要知道每位数字是什么，所以首先定义：

```
> digits2 :: Int -> (Int,Int)
> digits2 n = (div n 10, mod n 10)
```

数字 div n k 是 n 被 k 除的整数商，mod n k 是余数。也可以写成

```
digits2 n = (n `div` 10, n `mod` 10)
```

其中运算'div'和'mod'是 div 和 mod 的中缀形式，即运算写在运算数的中间，而不是写在运算数之前。这种形式非常便于改善可读性。例如，数学家会用 x div y 和 x mod y 表示这些表达式。注意，反引号（`）不同于表示单个字符的单引号（'）。

现在可以定义：

```
> convert2 :: Int -> String
> convert2 = combine2 . digits2
```

函数 combine2 的定义使用 Haskell 的*条件等式*（guarded equation）：

```
> combine2 :: (Int,Int) -> String
> combine2 (t,u)
>   | t==0          = units!!u
>   | t==1          = teens!!u
>   | 2<=t && u==0 = tens!!(t-2)
>   | 2<=t && u/=0 = tens!!(t-2) ++ "-" ++ units!!u
```

欲理解这段代码，需要知道 Haskell 表达等式和比较测试的如下符号：

9

==（等于）

/=（不等于）

<=（小于等于）

这些函数具有确切的类型，这将在稍后解释。

我们还需要知道两个测试的合取用 && 表示。因此，如果 a 和 b 都是 True，那么 a && b 返回布尔值 True，否则返回 False。实际上，有

```
(&&) :: Bool -> Bool -> Bool
```

第 2 章将进一步介绍类型 Bool。

最后，(++)表示两个列表串联的运算。该运算不关心列表的元素类型，所以

```
(++) :: [a] -> [a] -> [a]
```

例如，下列等式将两个函数（函数类型均为 Float -> Float）列表串联：

```
[sin,cos] ++ [tan] = [sin,cos,tan]
```

也可以串联两个字符列表：

```
"sin cos" ++ " tan" = "sin cos tan"
```

函数 combine2 的定义是在仔细考虑了所有可能的情况后得到的。稍加思考后可以看出，这里有 3 种主要情况，即十位数为 0、1 或者大于 1 的 3 种情况。对于前两种情况，答案是直接的，但是第 3 种情况需要划分为两种情况，即个位数是 0 或者非 0。这些情况的书写先后顺序，也就是这些条件等式的先后顺序并不重要，因为这些条件互不相交（两种情况不会同时为真），并且覆盖了所有的情况。

也可以定义：

```
combine2 :: (Int,Int) -> String
combine2 (t,u)
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
```

10

但是，这里书写等式的顺序很重要。条件的计算是自上而下的，并将第一个计算为 True 的条件对应的等式右边作为函数定义的结果。标识符 otherwise 是 True 的同义词，所以它涵盖了所有其他情况。

定义 convert2 的另一种方法：

```
convert2 :: Int -> String
convert2 n
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
  where (t,u) = (n `div` 10, n `mod` 10)
```

这里使用了 where 子句。这种子句引入了局部定义，其上下文或辖域是 convert2 定义的所有等式右边部分。这种局部定义对于定义的组织并使得定义可读性更强是非常重要

的。对于本例来说，where 子句避免了显式地定义函数 digits2。

以上定义相对简单。现在考虑函数 convert3，其参数 n 满足 $0 \leqslant n < 1000$，即 n 最多有 3 位数。其定义如下：

```
> convert3 :: Int -> String
> convert3 n
>   | h==0      = convert2 t
>   | t==0      = units!!h ++ " hundred"
>   | otherwise = units!!h ++ " hundred and " ++ convert2 t
>   where (h,t) = (n `div` 100, n `mod` 100)
```

使用这样的方式将数字分解，是因为可以使用 convert2 处理小于 100 的数字。

现在假定 n 满足 $0 \leqslant n < 1\,000\,000$，即 n 可以有 6 位数。沿用以上的模式，可以给出如下定义：

```
> convert6 :: Int -> String
> convert6 n
>   | m==0      = convert3 h
>   | t==0      = convert3 m ++ " thousand"
>   | otherwise = convert3 m ++ " thousand" ++ link h ++
>                 convert3 h
>   where (m,h) = (n `div` 1000,n `mod` 1000)
```

11

对于 $0 < m$ 且 $0 < h < 100$，表示 m 的词与表示 h 的词之间需要一个连接词“and”，所以定义：

```
> link :: Int -> String
> link h = if h < 100 then " and " else " "
```

该定义使用了条件表达式：

```
if <test> then <expr1> else <expr2>
```

也可以使用条件等式：

```
link h | h < 100   = " and "
       | otherwise = " "
```

根据不同情况，有时一种表达式可读性较强，有时另一种可读性更强。这里的 if、then 和 else，以及其他的一些词，称为 Haskell 保留字，这也意味着程序员不可以使用这些词做其他定义的名称。

注意函数 convert6 是如何使用简单的函数 convert3 来定义的，同时注意 convert3 是如何用更简单的函数 convert2 来定义的。这是函数定义的一般方法。在本例中，简单情况的考虑都派上了用场，因为最后函数的定义使用了简单情况的定义。

另外一点，把所求的函数命名为 convert6，但是开始时称该函数为 convert。没关系，可以定义：

```
> convert :: Int -> String
> convert = convert6
```

下面需要做的是将函数 convert 应用于一些输入参数。怎么做呢？

1.5 Haskell 平台

访问网页 www. haskell. org 可以看到如何下载 Haskell 平台（Haskell Platform）。该平台

是可用于运行 Haskell 脚本的工具和包的集合。平台包括支持 Windows、Mac 和 Linux 的 3 种版本，本书只介绍 Windows 版本，因为其他版本的用法相似。

平台中的一种工具是交互式计算器，称为 GHCi。它是 Glasgow Haskell Compiler Interpreter 的简称。计算器有 Windows 版本，称为 WinGHCi。打开该窗口将会看到下列信息： 12

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

这里的提示符 `Prelude >`表示包含预定义的函数、类型和其他值的标准库已经载入系统，现在 GHCi 可以用作超级计算器：

```
Prelude> 3^5
243
Prelude> import Data.Char
Prelude Data.Char> map toLower "HELLO WORLD!"
"hello world!"
Prelude Data.Char>
```

函数 `toLower` 在库 `Data.Char` 中定义。将该库输入后，用户便可以使用库中定义的函数了。注意提示符的变化，它显示了已经输入的库。这样的提示符很快会变得很长。但是，用户任何时候都可以修改提示符：

```
Prelude> :set prompt ghci>
ghci>
```

为简洁起见，本书将使用这种简单提示符。

用户可以输入一个脚本，如包含函数 `convert` 定义的 `Numbers2Words`：

```
ghci> :load "Numbers2Words.lhs"
[1 of 1] Compiling Main   ( Numbers2Words.lhs, interpreted )
Ok, modules loaded: Main.
ghci>
```

第 2 章将介绍模块的概念。例如，现在可以键入：

```
ghci> convert 301123
"three hundred and one thousand one hundred and twenty-three"
ghci>
```

13

本章最后以习题结束。这些习题包括其他有趣的知识，应该视为本章内容的有机组成部分。这也适用于后续的章节，所以，即使你不打算回答这些问题，也请阅读问题。习题答案附在习题后面。

1.6 习题

习题 A 考虑将一个整数加倍的函数：

```
double :: Integer -> Integer
double x = 2*x
```

下列表达式的值是什么？

```
map double [1,4,4,3]
map (double . double) [1,4,4,3]
map double []
```

假设 sum :: [Integer] -> Integer 是对一个整数列表求和的函数。下列哪些等式成立？为什么？

```
sum . map double = double . sum
sum . map sum    = sum . concat
sum . sort       = sum
```

读者需要回顾函数 concat 的功能。函数 sort 将一个数的列表按照递增顺序排序。

习题 B 在 Haskell 中，函数应用的优先级高于其他任何运算，所以，double 3 +4 等同于(double 3) +4，而不是 double (3 +4)。下列哪些式子是 $\sin^2\theta$ 在 Haskell 中的表示？(Haskell 的幂用“^”表示。)

```
sin^2 theta        sin theta^2      (sin theta)^2
```

如何用 Haskell 合式表达式表示 $\sin 2\theta/2\pi$？

习题 C 如前所见，一个字符，即类型为 Char 的元素，用单引号表示；一个串用双引号表示。特别是，串"Hello World!"不过是下面列表的简短表示。

14

```
['H','e','l','l','o',' ','W','o','r','l','d','!']
```

一般的列表用中括号和逗号表示。(顺便说明，小括号是圆形的，中括号是方形的，大括号是花的。) 因此，'H' 和"H"具有不同的类型。它们的类型是什么？2001 和"2001"的区别是什么？

运算 ++将两个列表串联。请化简下列式子：

```
[1,2,3] ++ [3,2,1]
"Hello" ++ " World!"
[1,2,3] ++ []
"Hello" ++ "" ++ "World!"
```

习题 D 在高频词一例中，首先每个字母被转换为小写，然后计算文本中的词。另一种方法是反过来，先计算文本中的词，然后将每个词中的每个字母转换为小写。第一种方法可以表示为 words . map toLower，请给出第二种方法的类似表达式。

习题 E 如果一个运算⊕满足 $x \oplus (y \oplus z) = (x \oplus y) \oplus z$，则称它是可结合的 (associative)。数的加法是可结合的吗？列表的串联可结合吗？函数的复合可结合吗？请给出一个不可结合的数的运算的例子。

如果一个元素 e 满足：等式 $x \oplus e = e \oplus x = x$ 对所有的 x 成立，则称 e 为⊕的单位元 (identity element)。请问加法、串联运算和函数复合的单位元各是什么？

习题 F 我妻子有一本书，书名为《EHT CDOORRSSW AAAGMNR ACDIINORTY》，书中包含下面这样的列表：

```
6-letter words
--------------
...
eginor: ignore,region
eginrr: ringer
eginrs: resign,signer,singer
...
```

15

没错，这是一本易位构词词典。易位构词的字母被排序，并将这些结果按照字典序存储。与每个易位构词相关的是用这些字母构成的英文单词。请描述如何设计一个函数：

```
anagrams :: Int -> [Word] -> String
```

使得 anagram n 可以从一个按照字典序排列的英文单词列表中恰好抽取出 n 个字母的词，并生成一个串，当串被显示时，结果是 n 个字母词的易位构词列表。不需要定义各种函数，只要给出这些函数的名和类型，并描述每个函数的功能即可。

习题 G 用一首歌结束本节习题：

```
One man went to mow
Went to mow a meadow
One man and his dog
Went to mow a meadow

Two men went to mow
Went to mow a meadow
Two men, one man and his dog
Went to mow a meadow

Three men went to mow
Went to mow a meadow
Three men, two men, one man and his dog
Went to mow a meadow
```

设计一个 Haskell 函数 song :: Int -> String，使得 song n 是 n 个人（n men）的歌词。假设 $n<10$。

要打印这首歌，可以键入：

```
ghci> putStrLn (song 5)
```

函数 putStrLn 将在第 2 章介绍。建议这样开始：

```
song n  = if n==0 then ""
          else song (n-1) ++ "\n" ++ verse n
verse n = line1 n ++ line2 n ++ line3 n ++ line4 n
```

16

这样可以用*递归*定义 song。

1.7 答案

习题 A 答案

```
map double [1,4,4,3]            = [2,8,8,6]
map (double . double) [1,4,4,3] = [4,16,16,12]
map double []                   = []
```

由此可知，[]表示空列表。

以下所有等式成立：

```
sum . map double = double . sum
sum . map sum    = sum . concat
sum . sort       = sum
```

事实上，3 个等式中每个均是以下 3 个更简单定律的结论。

```
a*(x+y) = a*x + a*y
x+(y+z) = (x+y)+z
x+y     = y+x
```

当然，现在还不清楚如何证明这些等式成立。（此外，为了避免混乱，使用打字机体

符号 = 表示两个表达式的相等。但是，数学符号 = 用于如 $\sin 2\theta = 2\sin\theta\cos\theta$ 的等式。）

习题 B 答案 表达式 `sin theta^2` 和 `(sin theta)^2` 都正确，但是 `sin ^2 theta` 是错误的。$\sin 2\theta/2\pi$ 在 Haskell 中的表示为

```
sin (2*theta) / (2*pi)
```

注意，如果写成

```
sin (2*theta) / 2 * pi = (sin (2*theta) / 2) * pi
```

则不符合要求。原因是/和* 这样的运算具有同样的优先级，并且是左结合的。第 2 章将
17 进一步讨论这些问题。

习题 C 答案

```
'H'    :: Char
"H"    :: [Char]
2001   :: Integer
"2001" :: [Char]
```

此外，`'\'` 表示转义字符，所以 `'\n'` 表示换行符，`'\t'` 表示制表符（tab 字符）。同样，`'\\'` 表示反斜杠，`"\\n"` 表示反斜杠和字母 n 这两个字符构成的列表。所以，文件路径 `C:\firefox\stuff` 用 Haskell 字符串表示成 `"C:\\firefox\\stuff"`。

```
[1,2,3] ++ [3,2,1]        = [1,2,3,3,2,1]
"Hello" ++ " World!"      = "Hello World!"
[1,2,3] ++ []             = [1,2,3]
"Hello" ++ "" ++"World!"  = "HelloWorld!"
```

如果你的最后两个答案正确，那么你就理解了 `[]` 表示任何对象的空列表，`""` 表示字符的空列表。

习题 D 答案 “将每个词中的每个字母转换为小写”包含了答案的线索。将一个词中每个字母转换为小写可表示成 `map toLower`，所以这个问题的答案是 `map (map toLower)`。这也表示下列等式成立：

```
words . map toLower = map (map toLower) . words
```

习题 E 答案 数值加法、列表串联和函数复合都是可结合的。当然，数的减法不可结合，幂运算也不可结合。加法的单位元是 0，串联的单位元是空列表，函数复合的单位元是下面定义的恒等函数：

```
id :: a -> a
id x = x
```

习题 F 答案 这个习题可仿照 3.1 节来解。一种定义函数 `anagrams n` 的方法如下：

1. 使用下列函数，抽取长度为 n 的词：

18

```
getWords :: Int -> [Word] -> [Word]
```

2. 给每个词添加一个标签。标签包含构成该词的字母，并按照字母序排列。例如，`word` 变成了标签和词的二元组 `("dorw","word")`。下列函数可以完成添加标签：

```
addLabel :: Word -> (Label,Word)
```

其中：

```
type Label = [Char]
```

3. 将标签词列表按照标签的字典序排序，该函数具有下列类型：

```
sortLabels :: [(Label,Word)] -> [(Label,Word)]
```

4. 将每组标签相同的连续相邻标签和词的二元组用一个新的二元组代替，这个新的二元组的第一个分量是它们的共同标签，第二个分量是具有这个相同标签的词的列表。该函数定义为

```
groupByLabel :: [(Label,Word)] -> [(Label,[Word])]
```

5. 利用下列函数将前面得到的每个二元组替换，并将结果串联在一起：

```
showEntry :: (Label,[Word]) -> String
```

最后得到

```
anagrams n = concat . map showEntry . groupByLabel .
             sortLabels . map addLabel . getWords n
```

习题 G 答案 一种可能解是

```
song n  = if n==0 then ""
          else song (n-1) ++ "\n" ++ verse n
verse n = line1 n ++ line2 n ++ line3 n ++ line4 n

line1 n = if n==1 then
          "One man went to mow\n"
          else
          numbers!!(n-2) ++ " men went to mow\n"
line2 n = "Went to mow a meadow\n"
line3 n = if n==1 then
          "One man and his dog\n"
          else
          numbers!!(n-2) ++ " men, " ++ count (n-2)
           ++ "one man and his dog\n"
line4 n = "Went to mow a meadow\n\n"

count n = if n==0 then ""
          else
          numbs!!(n-1) ++ " men, " ++ count (n-1)

numbers = ["Two", "Three", "Four", "Five", "Six",
           "Seven", "Eight", "Nine"]
numbs   = ["two", "three", "four", "five", "six",
           "seven", "eight"]
```


注意，脚本中省略了辅助函数和值的类型。尽管 Haskell 可以推导出所有函数和值的正确类型，但是，好的习惯是在脚本中说明它们的类型，即使这些类型很简单也要说明。脚本中明确的类型签名使得脚本更容易阅读，而且对于检查定义的合理性也很有帮助。

1.8 注记

如果读者对 Haskell 的起源感兴趣，那么一定要阅读《Haskell 的历史》(The History of Haskell)，可以从下列链接得到文章的拷贝：

research. microsoft. com/~ simonpj/papers/history-of-haskell

Haskell 的一个永恒的优势是它没有设计成一种封闭的语言，并鼓励研究人员通过扩展语言或者添加函数库尝试新的程序设计思想和技术。所以，Haskell 是一种大规模语言，读者可以找到关于 Haskell 各个方面的大量书籍、辅导教程和论文，包括最近由 Simon Marlow（O'Reilly，2013）编写的《Haskell 并行与并发程序设计》（Parallel and Concurrent Programming in Haskell）。

网页 www. haskell. org 包含大量有关资料的链接指引。不过，我在撰写本书时，桌上始终摊开着三本书。一本是由 Simon Peyton Jones 等编写的《Haskell 98 语言和库修订报告》（Haskell 98，Languages and Libraries，The Revised Report）（剑桥大学出版社，2003）。

报告对于理解 Haskell 第一标准版 Haskell 98 的本质是必不可少的。报告的在线版可以从下列链接获得：

20 www. haskell. org/onlinereport

本书大部分内容遵循标准版本，但是无论如何并未涵盖整个语言。

之后一个新版本 Haskell 2010 已经发布，参见

haskell. org/onlinereport/haskell2010/

其中的一个变化是模块名使用层次结构，例如，使用列表工具时调用模块 `Data.List`，而不是简单地写 `List`。

另外两本书是由 Bryan O'Sullivan、John Goeerzen 和 Don Stewart 编写的《真实世界的 Haskell》（Real World Haskell）（O'Reilly，2009），以及由 Graham Hutton 编写的《Haskell 程序设计》（剑桥大学出版社，2007）。一如其名，前一本书主要介绍非常实际的应用，后一本书是入门教程。Graham Hutton 曾笑着建议我将本书命名为《象牙塔 Haskell》。

关于高频词问题的历史非常有趣。Jon Bentley 曾请一位程序员 Don Knuth 编写一个高频词的文学 WEB 程序，然后请另一位程序员 Doug McIlroy 给程序一个文学评论，结果刊登在 Bentley 所撰写的文章“Programming Pearls”，见 Communications of the ACM. vol 29,

21 no. 6（June 1986）。

表达式、类型和值

Haskell 的每个合式（well-formed）表达式根据其定义都有一个合式类型，并且每个合式表达式根据其定义都有一个值（value）。对一个给定表达式求值时，步骤如下：

- GHCi 检查该表达式在语法上是否正确，即表达式是否符合 Haskell 的语法规则。
- 如果表达式语法正确，GHCi 推导出该表达式的类型，或者检查程序员指定的类型是否正确。
- 如果表达式类型正确，GHCi 对表达式求值：将表达式化简为最简单形式，即为其值。如果该值可打印，GHCi 将其显示在终端。

本章将仔细研究 Haskell 的求值过程。

2.1 GHCi 会话

检查一个表达式是否为合式的方法当然是使用 GHCi。GHCi 有一个命令：`type expr`，如果表达式 `expr` 是合式，则命令返回其类型。以下是一个 GHCi 会话（对某些 GHCi 返回信息做了简化）：

```
ghci> 3 +4)
<interactive>:1:5: parse error on input `)'
```

GHCi 在抱怨第 1 行第 5 个字符不符合系统的要求，换句话说，该表达式在语法上是错误的。


```
ghci> :type 3+4
3+4 :: Num a => a
```

GHCi 推断 3 +4 的类型是一个数值类型。稍后将对此做进一步解释。

```
ghci> :type if 1==0 then 'a' else "a"
<interactive>:1:23:
Couldn't match expected type `Char' with actual type `[Char]'
In the expression: "a"
In the expression: if 1 == 0 then 'a' else "a"
```

在一个条件表达式中：

```
if test then expr1 else expr2
```

GHCi 要求 `expr1` 和 `expr2` 的类型必须是相同的。但是，一个字符不是字符的列表。所以，尽管该条件语句符合 Haskell 语法，但它不是合式。

```
ghci> sin sin 0.5
<interactive>:1:1:
No instance for (Floating (a0 -> a0))
arising from a use of `sin'
Possible fix: add an instance declaration for
  (Floating (a0 -> a0))
In the expression: sin sin 0.5
In an equation for `it': it = sin sin 0.5
```

GHCi 返回一个很模糊的错误信息，说明该表达式不是合式。

```
ghci> sin (sin 0.5)
0.4612695550331807
```

啊哈，GHCi 对这个表达式很满意。

```
ghci> :type map
map :: (a -> b) -> [a] -> [b]
```

GHCi 返回该函数的类型。

```
ghci> map
<interactive>:1:1:
No instance for (Show ((a0 -> b0) -> [a0] -> [b0]))
arising from a use of `print'
Possible fix:
add an instance declaration for
  (Show ((a0 -> b0) -> [a0] -> [b0]))
In a stmt of an interactive GHCi command: print it
```

23

GHCi 表示不知道如何打印一个函数。

```
ghci> :type 1 `div` 0
1 `div` 0 :: Integral a => a
```

GHCi 表示 `1 `div` 0` 的类型是整型数。因此，`1 `div` 0` 是合式，并有一个值。

```
ghci> 1 `div` 0
*** Exception: divide by zero
```

GHCi 返回一个错误信息。那么 `1 `div` 0` 的值是什么？答案是，这是一个很特殊的值，数学上记作 ⊥，读作“bottom”。实际上，Haskell 给这个值提供了名称，不过不是 bottom，而是 `undefined`。

```
ghci> :type undefined
undefined :: a
ghci> undefined
*** Exception: Prelude.undefined
```

Haskell 不会显示 ⊥ 这个值。它可能返回一个错误信息，也可能在计算无限循环，因而一直保持沉默，直至用户中断该无穷计算过程。这种计算也可能引起 GHCi 崩溃。是的，没错。

```
ghci> x*x where x = 3
<interactive>:1:5: parse error on input `where'

ghci> let x = 3 in x*x
9
```

在 Haskell 中一个 `where` 子句不构成一个表达式，一个定义的等号右边的整个式子才是一个表达式。所以，以上第一个式子不是合式表达式。另一方面，在一个 `let` 表达式中：

```
let <defs> in <expr>
```

假定 `<defs>` 中的定义是合式，而且 `<expr>` 是合式，那么该 `let` 表达式是合式。`Let` 表达式在后面使用不多，但有的时候这种表达式很有用。

24

2.2 名称和运算符

如前所讲，一个脚本是一些名称及其定义的集合。函数和值的名以小写字母开头，但是数据构造器（稍后介绍）以大写字母开头。类型名（如 Int）、类族名（如 Num）和模块名（如 Prelude 和 Data.Char）也以大写字母开头。

一个运算符是一种出现在参数中间的特殊函数名，如 x + y 中的 +、xs ++ ys 中的 ++。运算符以符号开头。任何（非符号）二元函数都可以用反引号将其括起来，从而转换成一个运算符，而且任何运算符用括号括起来就变成了前缀名。例如：

```
3 + 4       等同于     (+) 3 4
div 3 4     等同于     3 `div` 4
```

运算符有不同的优先级（结合力）。例如：

```
3 * 4 + 2       等同于 (3 * 4) + 2
xs ++ yss !! 3  等同于 xs ++ (yss !! 3)
```

如有疑问，可使用括号消除歧义。另外，可以使用读者喜欢的任何名称来命名列表，包括 x、y、goodylist，等等。但是，一个便于记忆的简单规则是，用 x 表示对象，xs 表示对象的列表，xss 表示对象列表的列表。这也就解释了上面最后一行使用 yss 的原因。

优先级相同的运算符通常有一定的结合顺序，左结合或者右结合。例如，普通的算术运算是左结合的：

```
3 - 4 - 2  等同于 (3 - 4) - 2
3 - 4 + 2  等同于 (3 - 4) + 2
3 / 4 * 5  等同于 (3 / 4) * 5
```

函数应用运算具有最高优先级，而且是左结合的：

```
eee bah gum     等同于 (eee bah) gum
eee bah gum*2   等同于 ((eee bah) gum)*2
```

有些运算是右结合的：

```
(a -> b) -> [a] -> [b]  等同于 (a -> b) -> ([a] -> [b])
x ^ y ^ z               等同于 x ^ (y ^ z)
eee . bah . gum         等同于   eee . (bah . gum)
```

25

当然，如果一个运算满足结合律，如函数复合，那么结合的不同顺序对结果没有影响（值是一样的）。同样，仍然可以用括号消除可能产生的歧义。

也可以定义新的运算符，例如：

```
(+++) :: Int -> Int -> Int
x +++ y = if even x then y else x + y
```

条件表达式具有较低的优先级，所以，上述表达式等同于下列表达式：

```
if even x then y else (x + y)
```

而不是(if even x then y else x) + y。同样，仍然可以用括号表达不同的组合。

如果需要，也可以说明（+++）的优先级和结合顺序，但是这里不做介绍。

部分运算与兰姆达表达式

尽管这仅是一种风格，但是，大体来讲，人们更喜欢将所有辅助函数写在脚本中并显式命名。比如，如果需要给一个数加 1 的函数，或者将一个数加倍的函数，那么可以如下显式命名这些函数：

```
succ, double :: Integer -> Integer
succ n   = n+1
double n = 2*n
```

不过，Haskell 提供了命名这两个函数的其他方法，即(+1)和(2*)。这种机制称为部分运算（section）。部分运算中包括了运算符和部分参数。例如：

```
(+1) n = n+1
(0<) n = 0<n
(<0) n = n<0
(1/) x = 1/x
```

26

部分运算无疑是命名简单辅助函数的好方法，所以将其列入适量使用的好东西目录。

关于部分运算有一点重要附加说明：虽然(+1)表示加 1 的部分运算，但是(-1)并不是减 1 的部分运算，它只表示数值 -1。Haskell 用减号既表示二元减法运算，也表示取反的前缀运算。

假设欲定义一个函数：先将一个数加倍，然后在此基础上加 1。这个函数可以用两个部分运算的复合表示为(+1) . (*2)。但是，这个结果并不令人满意，因为表达式看起来太神秘，看到这个式子的读者必须停顿下来思考其含义。另一种方法似乎需要命名这个函数，但是，什么样的名合适呢？想出一个有意义的名称真难。

一种解决方法是使用兰姆达表达式（lambda expression）\n -> 2*n+1。在数学上这个函数写成 $\lambda n.\, 2n+1$，故称为兰姆达表达式。该表达式读作“n 的函数，其返回值是 $2n+1$”。例如：

```
ghci>  map (\n -> 2*n+1) [1..5]
[3,5,7,9,11]
```

有时兰姆达表达式似乎是描述一个函数的最好方法，但也仅仅是在某些不多的场合才使用这种表达式。

2.3 求值

Haskell 对一个表达式的求值过程是将其化简为最简形式，然后显示结果。例如，假设如下定义：

```
sqr :: Integer -> Integer
sqr x = x*x
```

基本上有两种方法将表达式 sqr (3+4)化简为最简形式，即 49。或者先对 3+4 求值，或者先使用 sqr 的定义：

```
  sqr (3+4)                  sqr (3+4)
= sqr 7                    = let x = 3+4 in x*x
= let x = 7 in x*x         = let x = 7 in x*x
= 7*7                      = 7*7
= 49                       = 49
```

27

两种方法的化简步数一样，但是化简的顺序稍有区别。左边的方法称为最内优先化简（innermost reduction），又称勤奋求值（eager evaluation）；右边的方法称为最外优先化简（outermost reduction）或者惰性求值（lazy evluation）。使用勤奋求值方法，总是先对参数求值，然后再应用函数。使用惰性求值方法，总是先带入函数定义，函数的参数求值只有在必要时才进行。

这两种方法是不是看似没多大区别？但是，考虑下面（稍做了简化）关于函数 fst 的求值过程，其中 fst 返回一个二元组的第一个分量，即 fst (x,y) = x。

```
  fst (sqr 1,sqr 2)          fst (sqr 1,sqr 2)
= fst (1*1,sqr 2)          = let p = (sqr 1,sqr 2)
= fst (1,sqr 2)              in fst p
= fst (1,2*2)              = sqr 1
= fst (1,4)                = 1*1
= 1                        = 1
```

需要注意到的关键区别是，勤奋求值方法对表达式 sqr 2 求值，但是惰性求值不需要该表达式参与，因此不会对其求值。

假设再给出下列定义：

```
infinity :: Integer
infinity = 1 + infinity

three :: Integer -> Integer
three x = 3
```

对 infinity 求值时，GHCi 将尝试计算 1 + (1 + (1 + (1 + (1 +...，因此陷入长久沉默，直至系统耗尽内存返回一个错误信息。这里 infinity 的值是⊥。

对 three infinity 求值同样有两种方法：

```
  three infinity                three infinity
= three (1+infinity)          = let x = infinity in 3
= three (1+(1+infinity))      = 3
= ...
```

在这里勤奋求值陷入试图对 infinity 求值的死循环中，但是惰性求值立即返回答案 3，求得结果 3 并不需要对 three 的参数求值。

另一个求值例子是阶乘函数的一种定义： 28

```
factorial :: Integer -> Integer
factorial n = fact (n,1)

fact :: (Integer,Integer) -> Integer
fact (x,y) = if x==0 then y else fact (x-1,x*y)
```

这是另一个递归定义（recursive defninition）的例子（函数 infinity 的定义以及第 1 章函数 song 的定义也都是递归的）。涉及递归函数的表达式的求值方法与其他函数的求值方法相同。

下面显示两种求值方法的化简步骤序列（为了说明问题，下面省略了条件表达式的化简步骤）：

```
  factorial 3                    factorial 3
= fact (3,1)                   = fact (3,1)
= fact (3-1,3*1)               = fact (3-1,3*1)
= fact (2,3)                   = fact (2-1,2*(3*1))
= fact (2-1,2*3)               = fact (1-1,1*(2*(3*1)))
= fact (1,6)                   = 1*(2*(3*1))
= fact (1-1,1*6)               = 1*(2*3)
= fact (0,6)                   = 1*6
= 6                            = 6
```

这里想说明的要点是，虽然两钟方法的化简步骤基本相同，但是惰性求值为了得到答案需要更大的空间，表达式 `1*(2*(3*1))` 在被求值前需要先在内存中构造起来。

惰性求值的优点是，只要任何一种化简顺序终止，惰性求值就会终止；它的化简步骤数永远小于勤奋求值步骤数，而且有时是及其地小。惰性求值的缺点是，它需要更多的空间，而且难于理解化简的准确次序。

Haskell 使用惰性求值。ML（另一种流行的函数语言）使用勤奋求值。习题 D 讨论为什么惰性计算是优点。第 7 章将进一步讨论惰性求值。

如果一个 Haskell 函数 `f` 满足 `f undefined = undefined`，则称 `f` 是*严格的*（strict），否则称为*非严格的*（non-strict）。函数 `three` 是非严格的，而 `(+)` 对于两个参数都是严格的。因为 Haskell 使用惰性求值，所以可以定义非严格的函数。这也是为什么 Haskell 被称为一种非严格的函数语言。

29

2.4 类型和类族

Haskell 定义了内置（或者初始）类型，如 `Int`、`Float` 和 `Char`。布尔值的类型 `Bool` 在标准引导库中定义：

```
data Bool = False | True
```

这是一个*数据声明*（data declaration）的例子。这里声明了类型 `Bool` 有两个*数据构造函数*：`False` 与 `True`。实际上类型 `Bool` 有 3 个值（而不是两个）：`False`、`True` 和 `undefined :: Bool`。为什么需要第三个值呢？考虑下列函数：

```
to :: Bool -> Bool
to b = not (to b)
```

引导库中 `not` 的定义如下：

```
not :: Bool -> Bool
not True  = False
not False = True
```

函数 `to` 的定义是完整的，但是对 `to True` 求值导致 GHCi 陷入无限循环，所以其值为类型 `Bool` 的 $\perp$。关于数据的声明，今后的章节会有更多介绍。

Haskell 包含内置的复合类型，例如：

```
[Int]             元素类型为Int的列表
(Int,Char)        一个Int和一个Char的序对
(Int,Char,Bool)   多元组类型
()                零元组类型
Int -> Int        一个由Int到Int的函数
```

类型()的唯一元素也用()表示。实际上，类型()还有第二个元素，即 undefined :: ()。现在明白了，每个类型都包含了值⊥。

如前所讲，在定义一个值或者函数时，好的习惯是同时在定义中说明它们的类型。

考虑下一个函数 take n，其作用是将列表的前 n 个元素构成的列表返回。该函数在第 1 章曾出现过。例如：

```
take 3 [1,2,3,4,5] = [1,2,3]
take 3 "category"  = "cat"
take 3 [sin,cos]   = [sin,cos]
```

30

应该给 take 赋予什么样的类型呢？该函数并不关心列表的元素类型是什么，所以函数 take 称为多态（polymorphic）函数，其类型记作

```
take :: Int -> [a] -> [a]
```

其中 a 是一个类型变量（type variable）。类型变量用小写字母开头。类型变量可以用任何类型替换。

类似的多态函数还有：

```
(++) :: [a] -> [a] -> [a]
map  :: (a -> b) -> [a] -> [b]
(.)  :: (b -> c) -> (a -> b) -> (a -> c)
```

最后一行是函数复合运算的多态类型声明。

那么运算(+)的类型是什么呢？以下是其类型的一些建议：

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
(+) :: a -> a -> a
```

前两个类型似乎太局限，而最后一个又显得过于一般。例如，两个函数不可以相加，两个字符或者两个布尔值也不可相加，至少没有明显的相加方法。

解决这个问题的方法是引入类族（type class）：

```
(+) :: Num a => a -> a -> a
```

这个声明断言(+)的类型为 a -> a -> a，其中 a 是任意数值类型。一个类族，如 Num，包含了一些命名的方法，如(+)，这些方法在类族的不同实例上可以有不同的定义。因此，类族提供了重载（overloaded）的函数，即同一个函数名在不同类型上可以有不同的定义。重载是另一种多态。

数值类型相当复杂，将会在第 3 章详细讨论，所以在这里介绍一个比较简单的类族：

```
class Eq a  where
  (==),(/=) :: a -> a -> Bool
  x /= y    = not (x == y)
```

这个定义引入了类族 Equality，该类族的成员可以使用它的方法，即同一个相等测试(==)和不相等测试(/=)。类族给出(/=)的缺省（default）定义，所以用户只需要给出(==)的定义。

31

一个类型要成为类族 Eq 的成员，必须定义一个实例（instance）。例如：

```
instance Eq Bool  where
  x == y  = if x then y else not y

instance Eq Person  where
  x == y  = (pin x == pin y)
```

如果 pin :: Person -> Pin，那么后一个实例的正确性需要 Eq Pin。当然，并非必须将 Person 定义成 Equality 俱乐部的成员，也可以使用下面的定义：

```
samePerson :: Person -> Person -> Bool
samePerson x y = (pin x == pin y)
```

但是，在程序中便不能用(==)代替 samePerson，除非定义 Person 是 Eq 的实例。

下面是另外两个类族 Ord 和 Show 的简化定义：

```
class (Eq a) => Ord a  where
  (<),(<=),(>=),(>) :: a -> a -> Bool
  x < y  = not (x >= y)
  x <= y = x == y || x < y
  x >= y = x == y || x > y
  x > y  = not (x <= y)

class Show a  where
  show :: a -> String
```

布尔运算(||)表示析取 a||b 为真，仅当 a 和 b 中至少有一个为真。这个运算可以如下定义：

```
(||) :: Bool -> Bool -> Bool
a || b = if a then True else b
```

类族 Ord 中方法的缺省定义相互依赖，所以，在定义该类族的任何实例时，必须至少给出一种方法的具体定义才可以打破这种循环依赖（不同的是，Eq 中只有(/=)有缺省定义）。类族 Ord 需要 Eq 作为它的*超类族*（superclass），因为它的4个比较方法的缺省定义使用了(==)。

类族 Show 用于显示结果。如果一个结果的类型不是 Show 的成员，那么 Haskell 不能显示这种计算结果。2.5 节将进一步解释这个类族。

32

2.5 打印值

先看一个谜题：

```
ghci> "Hello"++"\n"++ "young" ++"\n"++ "lovers"
"Hello\nyoung\nlovers"
```

噢，我们想要的是

```
Hello
young
lovers
```

为什么 Haskell 没有打印出这个结果？其原因是，Haskell 对一个合式表达式求值后，将 show 应用于该值，生成一个可以在终端打印的字符串。将 show 应用于一个值 v 生成一个字符串，该字符串打印时看上去恰好像 v。例如：

```
show 42        = "42"
show 42.3      = "42.3"
show 'a'       = "'a'"
show "hello\n" = "\"hello\\n\""
```

打印结果需要使用一个 Haskell 命令：

```
putStrLn :: String -> IO ()
```

类型 IO a 是一个表示输入和输出计算的特殊类型，当这种类型的命令被执行时，Haskell 与外部世界发生交互，最后返回类型为 a 的值。如果返回值不重要，如 putStrLn，则使用零元组的值()。

所以，Haskell 统一使用生成和输出（show-and-put）的策略打印值。因为前面的谜题中，要打印的值已经是一个串，所以可以省略生成步骤，直接打印输出：

```
ghci> putStrLn ("Hello ++"\n"++ "young" ++"\n"++ "lovers")
Hello
young
lovers
```

Haskell 提供许多其他的输入和输出命令，如读写文件命令、显示图形命令，等等。 33
这些命令的顺序必须正确书写，为此 Haskell 提供了一种特殊的记号，称为 do 记法。命令的内容是第 10 章的主题，下面仅介绍一点粗浅知识。

作为一个例子，考虑第 1 章的高频词问题，其中曾定义了函数：

```
commonWords :: Int -> String -> String
```

使得 commonWords n 可应用于一个文本串，然后返回文本中 n 个出现最多的词表，并用一个串表示之。下列程序从一个文件读出文本，然后将结果写入另一个文件。类型 FilePath 是字符列表的另一个同义词：

```
cwords :: Int -> FilePath -> FilePath -> IO()
cwords n infile outfile
   = do {text <- readFile infile;
         writeFile outfile (commonWords n text);
         putStrLn "cwords done!"}
```

例如，对下列式子求值：

```
ghci> cwords 100 "c:\\WarAndPeace" "c:\\Results"
```

在一个 Windows 平台上的结果是读出文件 c:\WarAndPeace，然后结果写入 c:\Results。程序还在终端打印一个信息。

以上定义中的两个分函数具有下列类型：

```
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

假设不想用交互的方式调用 cwords，而是想作为一个独立程序运行。方法是定义一个类型为 IO ()，标识为 main 的值。下面是这样的一个程序：

```
main
  = do {putStrLn "Take text from where:";
        infile <- getLine;
        putStrLn "How many words:";
```

```
    n <- getLine;
    putStrLn "Put results where:";
    outfile <- getLine;
    text <- readFile infile;
    writeFile outfile (commonWords (read n) text);
34  putStrLn "cwords done!" }
```

关于 read 的解释参见习题 H。假设高频词程序脚本存储在文件 cwords.lhs 中。现在可以用 GHC，即格拉斯哥 Haskell 编译器（Glasgow Haskell Compiler）进行编译：

```
$ ghc cwords.lhs
```

编译后的程序存储在文件 cwords.exe 中。在 Windows 下运行程序，可键入：

```
$ cwords
```

然后遵照程序提示进行。

2.6 模块

假设函数 commonWords 非常有用，我们想在其他脚本中也能够使用它。为此，需要将高频词脚本转变成一个模块（module）。首先，将脚本如下改写：

```
module CommonWords (commonWords) where
import Data.Char (toLower)
import Data.List (sort,words)
...
commonWords :: Int -> String -> String
...
```

模块声明关键字 module 后接模块名，而且模块名必须以大写字母开头。此外，该脚本必须存储在名为 CommonWords.lhs 的文件中，以便 Haskell 能够找到该模块（至少在使用文学编程的情况下，否则要命名为 CommonWords.hs）。模块名后是一个出口（export，或称输出）的列表，这些出口包括该模块输出给其他脚本的函数、类型和其他值。出口列表必须用圆括号括起来。这个例子中只输出一个函数 commonWords 给其他脚本。一个模块中的出口是该模块在其他模块中唯一可见的对象。如果省略出口列表和圆括号，那么模块中的所有对象都成为出口。

定义模块后，使用下面的声明可以将模块输入到其他脚本，然后用 GHC 编译模块。

```
import CommonWords (commonWords)
```

35 Haskell 模块有两个主要优点。一个优点是程序员可以把脚本中相关函数分在一个小组，形成独立模块，从而可以将脚本组织成适当大小的块。另一个优点是在编译模块中的函数被编译成了特定的机器代码，从而使得表达式化简更顺畅，所以函数的求值要快得多。GHCi 是一个解释器（interpreter），不是编译器；解释器对更接近于源语言 Haskell 的表达式内部形式求值。

2.7 Haskell 版面

do 记法的例子使用了花括号（{}）和分号（;），这些符号是显式版面（explicit layout）的例子。花括号和分号只是用来控制版面，除此之外在 Haskell 表达式中没有其他意义。

这些符号也可以用于其他位置：

```
roots :: (Float,Float,Float) -> (Float,Float)
roots (a,b,c)
 | a == 0       = error "not quadratic"
 | disc < 0     = error "complex roots"
 | otherwise    = ((-b-r)/e, (-b+r)/e)
 where {disc = b*b - 4*a*c; r = sqrt disc; e = 2*a}
```

这里的 where 子句显式地使用了花括号和分号，没有使用 Haskell 的版面规则。使用版面规则可以如下书写：

```
where disc = b*b - 4*a*c
      r    = sqrt disc
      e    = 2*a
```

但是，不可以这样书写：

```
where disc = b*b - 4*a*c
        r = sqrt disc
        e = 2*a
```

每当关键字 where 或者 do（以及 let）后的左花括号被省略时，版面（或者越位(offside)）规则便起作用了。此时，关键字后的式子，无论是在同一行还是下一行，其位置被系统记下。对于后面的每一行，如果进一步缩进，则认为是前一式子的延续；如果缩进相同，则认为是一个新表达式的开始；如果缩进比记下的更少，则这部分版面结束。至少越位规则大致如此。

越位规则说明了类族和实例声明中的缩进原因：

```
class Foo a  where
   我是类族声明的一部分
   我也是
现在类族声明结束。
```

如果对规则存疑，总是可以使用花括号和分号代替规则。事实上，越位规则在 do 记法中仍然可能引起混淆。所以，建议使用安全带，即花括号和分号。

确实，足球场上的越位规则是复杂的。

2.8 习题

习题 A 这是关于优先级的问题，来自英国卫报的 Chris Maslanka 的谜题版：

“2 加 2 的一半等于 2 还是等于 3?”

习题 B 下面的表达式中有些是语法不正确的；有些是语法正确的，但是不具有合理的类型。有些是合式。请识别出它们。如果是合式表达式，请给出适当的类型。假设 `double :: Int -> Int`。建议读者不使用计算机查看答案，但是如果使用的话，会看到一些奇怪的错误信息。

这些表达式如下：

```
[0,1)
double -3
double (-3)
double double 0
```

```
if 1==0 then 2==1
"++" == "+" ++ "+"
[(+),(-)]
[[],[[]],[[[]]]]
concat ["tea","for",'2']
concat ["tea","for","2"]
```

37

习题 C 在过去美好的日子里，作者写论文可以用这样的标题：

"The morphology of prex-an essay in meta-algorithmics"

但是，现在的杂志希望所有词用大写开头：

"The Morphology Of Prex-An Essay In Meta-algorithmics"

请设计一个函数 `modernise :: String -> String`，确保论文标题满足如上要求。下面先回答一些有帮助的问题：

1. 函数 `toLower :: Char -> Char` 是将字母转换为小写的函数。你认为标准引导库中将字母转换为大写的函数名是什么？

2. 函数 `word :: String -> [Word]` 曾在第 1 章使用过。你认为下列引导库函数的作用是什么？

```
unwords :: [Word] -> String
```

提示：如果下列方程中有一个成立的话，哪个成立？

```
words . unwords = id
unwords . words = id
```

3. 函数 `head :: [a] -> a` 返回非空列表的第一个元素，`tail :: [a] -> [a]` 返回一个列表去除第一个元素后的尾部列表。如果已知一个列表的第一个元素是 `x`，尾部是 `xs`，如何构造该列表？

习题 D Beaver 是勤奋求值器，Susan 是惰性求值器[⊖]。如果 `xs` 是长度为 n 的列表，在计算 `head (map f xs)` 时 Beaver 会计算 `f` 多少次？Susan 会计算多少次？Beaver 更喜欢 `head . map f` 的哪种形式？

函数 `filter p` 对一个列表过滤，返回满足布尔条件测试 `p` 的元素构成的列表。`filter` 的类型为

```
filter :: (a -> Bool) -> [a] -> [a]
```

Susan 喜欢用函数 `head . filter p` 找出列表中第一个满足 `p` 的元素。为什么 Berver

38 不用同样的表达式？相反，Beaver 可能定义这样的函数：

```
first :: (a -> Bool) -> [a] -> a
first p xs | null xs   = error "Empty list"
           | p x       = ...
           | otherwise = ...
           where x = head xs
```

函数 `null` 对于空列表返回 `True`，非空列表返回 `False`。对 Beaver 的函数求值时，表达式 `error message` 终止求值过程，并在终端打印串 `message`，所以结果是 $\perp$。请完成 Beaver 定义中右边部分。

⊖ 如果读者不清楚这里的含义，请谷歌"lazy susan"查看其含义。

Beaver 可能更喜欢 head . filter p . map f 的哪种形式?

习题 E 类型 Maybe 在标准引导库中定义如下:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

该定义使用了一个 deriving（导出）子句。对于某些数据声明，Haskell 可以为它们自动生成一些标准类族的实例。对于目前的例子，导出子句意味着程序员无需键入下列冗繁的定义:

```
instance (Eq a) => Eq (Maybe a)
  Nothing == Nothing = True
  Nothing == Just y  = False
  Just x == Nothing  = False
  Just x == Just y   = (x == y)

instance (Ord a) => Ord (Maybe a)
  Nothing <= Nothing = True
  Nothing <= Just y  = True
  Just x <= Nothing  = False
  Just x <= Just y   = (x <= y)
```

定义中 Nothing 小于 Just y 的原因只是因为在 Maybe 的数据声明中，构造函数 Nothing 写在 Just 之前。

Maybe 类型提供了处理失败的系统方法，所以这个类型非常有用。再考虑前面习题中的函数:

39

```
first p = head . filter p
```

勤奋的 Beaver 和懒散的 Susan 都有该函数各自的版本，而且当把 first p 应用于一个空列表时都终止执行，并返回一个错误信息。这个结果并不圆满。更好的方法是定义:

```
first :: (a -> Bool) -> [a] -> Maybe a
```

现在如果列表中没有元素满足测试条件，可以通过返回 Nothing 很圆满地处理失败的情况。

请给出这个 first 版本的定义。

最后，计算 Haskell 中类型为 Maybe a ->Maybe a 的函数个数。

习题 F 下面是一个计算 x 的 n 次方的函数，其中 $n \geqslant 0$。

```
exp :: Integer -> Integer -> Integer
exp x n | n == 0    = 1
        | n == 1    = x
        | otherwise = x*exp x (n-1)
```

请问计算 exp x n 需要做多少次乘法?

一个聪明的程序员 Dick 声称他可以用次数少得多的方法计算 exp x n:

```
exp x n | n == 0  = 1
        | n == 1  = x
        | even n  = ...
        | odd n   = ...
```

请完成定义，并计算用 Dick 的方法对表达式 exp x n 求值需要做多少次乘法，这里

假定 $2^p \leqslant n < 2^{p+1}$。

习题 G 假设日期用 3 个整数表示成（day，month，year）。请定义一个函数 showDate :: Date -> String 使得日期能够像下面例子一样显示：

```
showDate (10,12,2013) = "10th December, 2013"
showDate (21,11,2020) = "21st November, 2020"
```

40 读者应该明白 Int 是类族 Show 的实例，所以 show n 生成十进制整数 n 的串表示。

习题 H 信用卡公司 Foxy 发行卡号（CIN）为 10 位数字的信用卡，前 8 位是任意的，但是最后两位表示的整数是前 8 位数字之和的验证码。例如，“6324513428”是一个合法的 CIN，因为前 8 位数字之和是 28。

请构造一个函数 addSum :: CIN -> CIN，它将 8 位数字组成的串转换为包含其验证码的 10 位数字的串。因此，CIN 是 String 的同义词，不过字符限制为数字。注意，Haskell 类型同义词不能强制这样的类型约束。读者需要实现一个数字字符和对应数值之间的转换。其中一个方向的转换很简单：只需使用 show。另一个方向的转换也很简单：

```
getDigit :: Char -> Int
getDigit c = read [c]
```

函数 read 是类族 Read 的方法，其类型为

```
read :: Read a => String -> a
```

类族 Read 是 Show 的对偶，read 也是 show 的对偶。例如：

```
ghci> read "123" :: Int
123
ghci> read "123" :: Float
123.0
```

使用函数 read 时必须提供结果的类型。任何时候都可以用这种方法给表达式添加类型注释（type annotation）。

现在请构造一个函数 valid :: CIN ->Bool 检查一个卡号是否合法。函数 take 在这里可能有帮助。

习题 I 根据定义，一个回文（palindrome）是这样的字符串：如果忽略标点符号、空格和字母的大小写，那么正念和反念结果是一样的。请设计一个交互式程序：

```
palindrome :: IO ()
```

运行该程序将引导出一个交互会话，例如：

41

```
ghci> palindrome
Enter a string:
Madam, I'm Adam
Yes!

ghci> palindrome
Enter a string:
A Man, a plan, a canal - Suez!
No!

ghci> palindrome
Enter a string:
Doc, note I dissent. A fast never prevents a fatness.
I diet on cod.
Yes!
```

其中函数 isAlpha :: Char -> Bool 检测一个字符是否为字母，函数 reverse :: [a] -> [a]将列表反转。函数 reverse 在标准引导库中定义，函数 isAlpha 可以从库 Data.Char 中输入。

2.9 答案

习题 A 答案 Maslanka 谜题的答案是“可能等于 2 也可能等于 3”。这个小谜题难倒了不少杰出的计算机科学家。

习题 B 答案 GHCi 会话显示（附加解释）：

```
ghci> :type [0,1)
<interactive>:1:5: parse error on input `)'
```

虽然 GHCi 没有聪明地指出应该使用']'，但是它明白')'是错误的。

```
ghci> :type double -3
<interactive>:1:9:
No instance for (Num (Int -> Int))
arising from the literal `3'
Possible fix: add an instance declaration for
  (Num (Int -> Int))
In the second argument of `(-)', namely `3'
In the expression: double - 3
```

42

错误信息解释说，数值减法(-)的类型是 Num a => a -> a。若要使得 double - 3 是合法表达式（题目中写的是 double -3，但是空格在这里不重要），double 必须是一个数，所以需要类族实例 Num (Int -> Int)。但是，不存在这样的实例定义，所以一个函数减去一个数无意义。

```
ghci> double (-3)
-6
ghci> double double 0
<interactive>:1:1:
The function `double' is applied to two arguments,
but its type `Int -> Int' has only one
In the expression: double double 0
In an equation for `it': it = double double 0
```

大多 GHCi 错误信息的含义是清楚的。

```
ghci> if 1==0 then 2==1

<interactive>:1:18:
parse error (possibly incorrect indentation)
```

条件表达式缺少 else 子句，因此不完整。

```
ghci> "++" == "+" ++ "+"
True
```

两边都是合式表达式，而且表示同一个列表。

```
ghci> [(+),(-)]
<interactive>:1:1:
No instance for (Show (a0 -> a0 -> a0))
arising from a use of `print'
```

```
Possible fix:
add an instance declaration for
  (Show (a0 -> a0 -> a0))
In a stmt of an interactive GHCi command: print it
```

要显示值[(+),(-)]，首先要能够显示其元素。但是，不存在显示函数的方法。

43

```
ghci> :type [[],[[]],[[[]]]]
[[],[[]],[[[]]]] :: [[[[a]]]]
```

为了解释这个类型，先假定主列表的类型是[b]。主列表的第一个元素是列表，所以b = [c]；第二个元素是列表的列表，故c = [d]；第三个元素是列表的列表的列表，故d = [a]。

```
ghci> concat ["tea","for",'2']
<interactive>:1:21:
Couldn't match expected type `[Char]'
with actual type `Char'
In the expression: '2'
In the first argument of `concat',
namely `["tea", "for", '2']'
In the expression: concat ["tea", "for", '2']
```

列表的前两个元素具有类型[Char]，但是最后一个元素的类型是Char，所以这种列表是不合法的。

```
ghci> concat ["tea","for","2"]
"teafor2"
```

习题C答案

1. 当然是toUpper。

2. 将词串接，并在词之间加一个空格。另外，word . unword = id成立，但是unword . word = id不成立。

3. [x] ++ xs。

```
modernise :: String -> String
modernise = unwords . map capitalise . words

capitalise :: Word -> Word
capitalise xs = [toUpper (head xs)] ++ tail xs
```

第4章将给出capitalise的另一种定义。

习题D答案 使用勤奋求值方法计算head (map f xs)需要对f计算n次，但是使用惰性求值只需对f计算一次。Beaver必须使用恒等式head . map f = f . head。

44

Beaver不使用定义first p = head . filter p，而是可能定义：

```
first :: (a -> Bool) -> [a] -> a
first p xs | null xs   = error "Empty list"
           | p x       = x
           | otherwise = first p (tail xs)
           where x = head xs
```

Beaver不使用定义first p f = head . filter p . map f，而是可能定义：

```
first :: (b -> Bool) -> (a -> b) -> [a] -> b
first p f xs | null xs   = error "Empty list"
```

```
        | p x       = x
        | otherwise = first p f (tail xs)
        where x = f (head xs)
```

问题的关键是，用勤奋求值策略时，大多数函数必须用显式递归定义，而不是利用像 map 和 filter 这样的函数定义。

习题 E 答案 懒散的 Susan 可能会定义：

```
first p xs = if null ys then Nothing
             else Just (head ys)
             where ys = filter p xs
```

类型为 Maybe a -> Maybe a 的函数共有 4 个：

```
f1 x = case x of Nothing -> Nothing; Just v -> Just v
f2 x = case x of Just v -> Just v
f3 x = case x of Nothing -> Just undefined; Just v -> Just v
f4 x = Just (case x of Nothing -> undefined; Just v -> v)
```

习题 F 答案 对 exp x n 求值需要 $n-1$ 次乘法。Dick 的方法是利用恒等式 $x^{2m}=(x^2)^m$ 和 $x^{2m+1}=x(x^2)^m$ 得到递归定义：

```
exp x n
    | n == 0  = 1
    | n == 1  = x
    | even n  = exp (x*x) m
    | odd n   = x*exp (x*x) m
    where m = n `div` 2
```

45

这是*分治法*（divide and conquer）算法的一个例子。使用 Dick 的方法计算 exp x n 最多需要 $2\lfloor \log n \rfloor$ 次乘法，其中 $\lfloor x \rfloor$ 表示一个数的底，即不大于该数的最大整数。第 3 章将进一步讨论取底函数。

习题 G 答案

```
showDate :: Date -> String
showDate (d,m,y) = show d ++ suffix d ++ " " ++
                   months !! (m-1) ++ ", " ++ show y
```

函数 suffix 计算数字的右后缀：

```
suffix d = if d==1 || d==21 || d==31 then "st" else
           if d==2 || d==22 then "nd" else
           if d==3 || d==23 then "rd" else
           "th"

months = ["January",.......]
```

如果读者乐于找到 suffix 的聪明计算法，那么读者应该明白：有时候简单的解就是最好的解。

习题 H 答案 一个解是

```
addSum :: CIN -> CIN
addSum cin =
  cin ++ show (n `div` 10) ++ show (n `mod` 10)
  where n = sum (map fromDigit cin)

valid :: CIN -> Bool
valid cin = cin == addSum (take 8 cin)
```

```
fromDigit :: Char -> Int
fromDigit c = read [c]
```

函数 fromDigit 返回一个字符的对应数字。

46 **习题 I 答案** 下面是一个解：

```
import Data.Char (toLower,isAlpha)

palindrome :: IO()
palindrome
 = do {putStrLn "Enter a string:";
       xs <- getLine;
       if isPalindrome xs then putStrLn "Yes!"
       else putStrLn "No!"}

isPalindrome :: String -> Bool
isPalindrome xs = (ys == reverse ys)
  where ys = map toLower (filter isAlpha xs)
```

2.10 注记

本章多次提到 Haskell “标准引导库”，该库包含许多程序设计不可缺少的基本的类型、类族、函数和值。关于标准引导库的完整说明，请参看 Haskell 报告第 8 章；或者访问下列链接：

www. haskell. org/onlinereport/standard-prelude. html

关于函数程序语言，特别是 Haskell 的实现，更多信息参看 www. haskell. org。早期由 Simon Peyton Jones 编写的《The Implementation of Functional Programming languages》（Prentice Hall，1987）一书已经不再版，但是可以从下列链接找到在线版：

research. microsoft. com/~ simonpj/papers/slpj-book-1987

除 GHC 外，Haskell 还有其他保持维护的编译器，包括 UHC（Utrecht Haskell Compiler），参见官网 cs. uu. nl/wiki/UHC。

关于勤奋求值和惰性求值的比较，请参考 Bob Harper 的博客文章“The point of laziness”，可以在下列链接找到：

existentialtype. wordpress. com/2011/04/24/

在博文中 Harper 列举了他偏爱严格语言的一些理由。也请阅读 Lennart Augustsson 的回复。Augustsson 的主要观点是，如习题 D 中所强调，在严格求值策略下，为了提高效
47 率，大部分函数必须用显式递归的方式定义，这样便失去了使用简单标准函数定义新函数的能力。这样削弱了使用组成函数的通用定律对函数进行推理的能力。

Bob Harper 是《The Definition of Standard ML（Revised）》（MIT Press，1989）的作者之一。ML 是一种严格语言。读者可以在下列链接找到有关 ML 的介绍：

www. cs. cmu. edu/ ~ rwh/smlbook/book. pdf

另一种越来越受欢迎的语言是 Agda，它既是一种依存类型函数语言，也是一个证明辅助器，请参见 Agda 官网：

wiki. portal. chalmers. se/agda/pmwiki. php

48 Chris Maslanka 是英国《卫报》的周六版专栏作家。

数

Haskell 中的数比较复杂，因为 Haskell 世界有许多不同类型的数，包括：

Int	有限精度整数，至少包含范围 $[-2^{29}, 2^{29})$，整数溢出不被检测
Integer	任意精度整数
Rational	任意精度有理数
Float	单精度浮点数
Double	双精度浮点数
Complex	复数（定义在 Data.Complex 中）

大多数程序以各种方式使用数，所以必须理解 Haskell 提供了哪些数，以及这些不同类型的数如何转换。这便是本章讨论的内容。

3.1 类族 Num

Haskell 的所有数都是类族 Num 的实例：

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

类族 Num 是类族 Eq 和 Show 的子类族。这表示每个数都可以被打印，任意两个数都可以比较是否相等，任意数都可以与另一个数值相加、相减和相乘，任意数都可以取反。
Haskell 允许用 -x 表示 negate x，这也是 Haskell 中唯一的一个前缀运算符。

函数 abs 和 signum 返回一个数的绝对值和符号。如果允许在 Num 中有次序运算（实际上在 Num 中没有次序运算，因为复数不能排序），那么就可以定义：

```
abs x    = if x < 0 then -x else x
signum x | x < 0  = -1
         | x == 0 = 0
         | x > 0  = 1
```

函数 fromInteger 是一个转换函数。一个整数值如 42 表示 fromInteger 在类型 Integer 的一个适当值上的应用，所以这样的数字具有类型 Num a => a。待下面介绍了数的其他类族和类族之间的转换后再来介绍这种选择。

3.2 其他数值类族

类族 Num 有两个子类族——实数和分数：

```
class (Num a,Ord a) => Real a where
  toRational :: a -> Rational
```

```
class (Num a) => Fractional a where
  (/) :: a -> a -> a
  fromRational :: Rational -> a
```

实数可以比较大小。类族 Real 除了从超类族 Ord 继承的比较运算外，只有一个新的方法，即该类族元素到 Rational 元素的转换函数。类型 Rational 本质上是整数对的同义词。实数 π 不是有理数，所以 toRational 只能将其转换成一个近似的有理数：

```
ghci>  toRational pi
884279719003555 % 281474976710656
```

50 并不像 22 % 7 那样好记，但是更精确。符号 % 用来分离一个有理数的分子和分母。

分数是定义了除法的数的集合。一个复数不可能是实数，但可以是分数。一个浮点数如 3.149 表示 fromRational 在某个适当的有理数上的应用。所以

```
3.149 ::  Fractional a => a
```

这个类型和前面的 42 的类型 Num a => a 解释了为什么可以构造像 42 + 3.149 这样一个整数和一个浮点数相加的合法表达式。两个类型都是类族 Num 的成员，而且所有的数都可以相加。考虑

```
ghci> :type 42 + 3.149
42 + 3.149 :: Fractional a => a
```

这表示相加的结果还是一个分数。

实数的一个子类族是整型数，该类族的简化定义如下：

```
class (Real a, Enum a) => Integral a where
  divMod :: a -> a -> (a,a)
  toInteger :: a -> Integer
```

类族 Integral 是 Enum 的一个子类族，而 Enum 的类型是其元素可以顺序枚举的类型。每个整型数可以用 toInteger 转换为一个 Integer。这也说明可以用两步将一个整型数转换为任意其他类型：

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

运算 divMod 返回两个值：

```
x `div` y  = fst (x `divMod` y)
x `mod` y  = snd (x `divMod` y)
```

标准引导库函数 fst 和 snd 返回一个二元组的第一个分量和第二个分量：

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

51

数学上，x div $y = \lfloor x/y \rfloor$。将在 3.3 节看到如何计算 $\lfloor x \rfloor$。x mod y 的定义如下：

$$x = (x \text{ div } y) * y + x \text{ mod } y$$

对于整数 x 和 y，有 0 <= x 'mod' y < y。

回顾第 1 章的函数 digits2，当时定义了

```
digits2 n = (n `div` 10, n `mod` 10)
```

更高效的定义是 digits2 n = n `divMod` 10，因为这里只需要执行一次 divMod 运算。更简洁点，可以利用部分运算，写成 digits2 = (`divMod` 10)。

Haskell 还有其他的数值类族，包括 Frational 的子类族 Floating，其中包含对数函数和三角函数。但是，这些已经够用了。

3.3 取底函数的计算

值 $\lfloor x \rfloor$ 表示 x 的底（floor），定义为满足 $m \leqslant x$ 的最大整数 m。定义一个函数 floor :: Float -> Integer 计算数的底。Haskell 在标准引导库中提供了这个函数，不过给出我们自己的定义有指导意义。

一个叫 Clever Dick（聪明的迪克）的学生拿到问题后给出下面的解：

```
floor :: Float -> Integer
floor = read . takeWhile (/= '.') . show
```

用语言来叙述：将输入的数显示成串，截取小数点前的字串，然后将结果读成一个整数。我们还没有遇到 takeWhile，显然 Clever Dick 知道这个函数。Clever Dick 的解在一些情况下是错的，习题 D 要求读者列出这些情况。

我们将借用一个显式的查找求一个数的底，为此需要一个循环：

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if p x then x else until p f (f x)
```

函数 until 也是由标准引导库提供的。下面是一个例子： 52

```
ghci>  until (>100) (*7) 1
343
```

本质上 until f p x 计算下列无穷列表中满足 p y = True 的第一个元素：

```
[x, f x, f (f x), f (f (f x)), ...]
```

关于 until 的精确解释参见第 4 章。

现在考虑 floor 的设计，不禁想分情况考虑，区别 $x < 0$ 和 $x \geqslant 0$ 两种情况。对于 $x < 0$，需要在序列 -1，-2，…中找到满足 $m \leqslant x$ 的第一个整数 m。由此得到参数为负数时的定义：

```
floor x = until (`leq` x) (subtract 1) (-1)
          where m `leq` x = fromInteger m <= x
```

这个定义中有几点需要注意。第一，引导库函数 subtract 的使用，其定义为

```
subtract x y = y-x
```

必须使用 subtract 1，因为（-1）不是部分应用，而是数 -1(until 的第三个参数)。

第二，为什么使用了`leq`而没有使用看似完全适合的(<=)？答案是(<=)具有下列类型：

```
(<=) :: Ord a => a -> a -> Bool
```

特别是，(<=)的两个参数具有相同的类型。但是想要的是

```
leq :: Integer -> Float -> Bool
```

其中两个参数具有不同的数值类型。因此，需要使用 fromInteger 将整数转换为浮点数。搞懂在某些情况下需要转换函数是理解 Haskell 算术的关键点。

第三，注意(`leq` x)与(leq x)不同：

```
(leq x)   y = leq x y
(`leq` x) y = y `leq` x = leq y x
```

这里容易出错。

53 如果不喜欢辅助定义，那么总是可以写成

```
floor x = until ((<=x) . fromInteger) (subtract 1) (-1)
```

在这个定义中将(`leq` x)定义为内联（inlined）。

接下来需要处理 $x \geqslant 0$ 的情况。对于这种情况，需要找出满足 $x < n+1$ 的第一个整数 n。可以先求出满足 $x < n$ 的第一个整数 n，然后再减去 1。由此得到定义：

```
floor x = until (x `lt` ) (+1) 1 - 1
          where x `lt` n = x < fromInteger n
```

将两个定义合在一起，得到

```
floor x = if x < 0
          then until (`leq` x) (subtract 1) (-1)
          else until (x `lt`) (+1) 1 - 1
```

（问题：为什么在第一行不必写成 x < fromInteger 0？）这个定义除了难看的分情况以及两种情况不对称外，真正的问题是效率很低：计算结果大致需要 $|x|$ 步（$|x|$ 是 abs x 的数学表示）。

二分查找

计算 floor 的更好方法是首先找到满足 $m \leqslant x < n$ 的整数 m 和 n，然后将区间（m, n）缩减成包含 x 的单位区间（满足 $m+1=n$ 的区间）。此时，返回区间的左边界作为结果。由此得到

```
floor :: Float -> Integer
floor x = fst (until unit (shrink x) (bound x))
          where unit (m,n) = (m+1 == n)
```

其中的值 bound x 是满足 $m \leqslant x < n$ 的一个二元组（m, n）。如果区间（m, n）不是单位区间，则 shrink x (m,n)返回一个严格小于原区间并仍然包含 x 的新区间。

首先考虑如何缩减一个包含 x 的非单位区间，即 $m \leqslant x < n$。假设 p 是满足 $m < p < n$ 的任意整数。这样的 p 一定存在，因为（m, n）不是单位区间。然后定义：

```
type Interval = (Integer,Integer)
```

54

```
shrink :: Float -> Interval -> Interval
shrink x (m,n) = if p `leq` x then (p,n) else (m,p)
                 where p = choose (m,n)
```

如何定义 choose 呢？两种选择是 choose (m,n) = m+1 和 choose (m,n) = n-1，

因为两者都减小区间长度。但是，更好的选择是

```
choose :: Interval -> Integer
choose (m,n) = (m+n) `div` 2
```

使用这个选择，每一步区间的长度减半，而不是减1。但是，需要检查在 $m+1 \neq n$ 的情况下 $m < (m+n)\,\mathrm{div}\,2 < n$ 成立。推理如下：

$$
\begin{aligned}
& m < (m+n)\,\mathrm{div}\,2 < n \\
\equiv & \quad \{\text{整数的序关系}\} \\
& m+1 \leqslant (m+n)\,\mathrm{div}\,2 < n \\
\equiv & \quad \{\text{因为}(m+n)\,\mathrm{div}\,2 = \lfloor (m+n)/2 \rfloor\} \\
& m+1 \leqslant (m+n)/2 < n \\
\equiv & \quad \{\text{算术}\} \\
& m+2 \leqslant n \wedge m < n \\
\equiv & \quad \{\text{算术}\} \\
& m+1 < n
\end{aligned}
$$

最后，如何定义 bound 呢？可以先定义：

```
bound :: Float -> Interval
bound x = (lower x, upper x)
```

其中值 lower x 是小于或者等于 x 的某个整数，upper x 是大于 x 的某个整数。比通过线性搜索查找这些值更好的搜索方法是

```
lower :: Float -> Integer
lower x = until (`leq` x) (*2) (-1)

upper :: Float -> Integer
upper x = until (x `lt`) (*2) 1
```

55

为了使得 bound 高效，更好的方法是每步加倍，而不是仅加1或者减1。例如，对于 $x = 17.3$，仅需要7次比较即可找到包围区间 $(-1, 32)$，然后用5步缩减到 $(17, 18)$。实际上，计算上界和下界总共需要的步数正比于 $\log|x|$，整个算法最多两倍于这个时间。一个对数时间算法比线性时间算法要快得多。

标准引导库用下列方法定义 floor：

```
floor x = if r < 0 then n-1 else n
          where (n,r) = properFraction x
```

其中函数 properFraction 是类族 RealFrac（目前还没有讨论过的类族，其方法处理数的舍入）的一个方法，它将一个数 x 分解为整数部分 n 和小数部分 r，使得 $x = n + r$。现在大家明白了。

3.4 自然数

Haskell 没有提供自然数的类型，即非负整数的类型。但是，我们自己可以定义这样的类型：

```
data Nat = Zero | Succ Nat
```

这是一个*数据声明*（data declaraion）的例子。声明表示，Zero 是 Nat 的一个值，而且当 n 是 Nat 的一个值时，Succ n 也是 Nat 的值。Zero 和 Succ 都称为*数据构造器*（data constructor）或构造函数，而且要用大写字母开头。Zero 的类型是 Nat，Succ 的类型是 Nat -> Nat。因此，下列每一个元素都是 Nat 的元素：

```
Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero))
```

下面考虑如何通过将 Nat 设置为类族 Num 的成员来进行算术运算编程。首先，需要将 Nat 设置为 Eq 和 Show 的实例：

```
instance Eq Nat where
 Zero == Zero     = True
 Zero == Succ n   = False
 Succ m == Zero   = False
 Succ m == Succ n = (m == n)
instance Show Nat where
 show Zero           = "Zero"
 show (Succ Zero)    = "Succ Zero"
 show (Succ (Succ n)) = "Succ (" ++ show (Succ n) ++ ")"
```

56

这些定义使用了*模式匹配*（pattern matching）。特别是，show 的定义利用了 3 个模式：Zero、Succ Zero 和 Succ (Succ n)。这些模式互不相同，而且覆盖了 Nat 除⊥之外的所有元素。

或者，也可以使用下列方法达到同样目的：

```
data Nat = Zero | Succ Nat  deriving (Eq,Ord,Show)
```

如第 2 章习题 E 所讲，聪明的 Haskell 可以自动生成某些标准类族的实例，包括 Eq、Ord 和 Show。

现在可以定义 Nat 为数值类型：

```
instance Num Nat where
  m + Zero     = m
  m + Succ n   = Succ (m+n)

  m * Zero     = Zero
  m * (Succ n) = m * n + m

  abs n           = n
  signum Zero     = Zero
  signum (Succ n) = Succ Zero

  m - Zero        = m
  Zero - Succ n   = Zero
  Succ m - Succ n = m - n

  fromInteger x
   | x <= 0    = Zero
   | otherwise = Succ (fromInteger (x-1))
```

我们定义了减法运算：如果 $m \leqslant n$，则 $m-n=0$。当然，Nat 上的算术运算极其慢，而且每个数都占用很大的空间。

57

非完整数

每个类型都包含值⊥。因此，对于所有类型 a 有 undefined :: a。根据定义 Succ

是一个非严格函数，下列这些值互不相同，而且都是 Nat 的元素：

```
undefined, Succ undefined, Succ (Succ undefined), ...
```

老实说，这些非完整数不是很有用，但是它们确实存在。可以把 Succ undefined 理解为这样的数，只知道这个数至少是 1：

```
ghci> Zero == Succ undefined
False
ghci> Succ Zero == Succ undefined
*** Exception: Prelude.undefined
```

Nat 还有另外一个数：

```
infinity :: Nat
infinity = Succ infinity
```

因此

```
ghci> Zero == infinity
False
ghci> Succ Zero == infinity
False
```

等等。

总之，Nat 的元素构成包括有穷数、非完整数和无穷数（只有一个）。对于其他数据类型有同样的结论：该类型包含有穷元素、非完整元素和无穷元素。

也可以选择 Succ 为严格的，通过下列方式达到目的：

```
data Nat = Zero | Succ !Nat
```

其中标记！称为严格标志（strictness flag）。对于这样的声明，有如下结果：

```
ghci> Zero == Succ undefined
*** Exception: Prelude.undefined
```

58

这次对等式测试求值迫使两边求值，对 Succ undefined 求值引起错误信息。定义 Succ 为严格的构造函数使得自然数只包含有穷数和一个无定义数。

3.5 习题

习题 A 以下哪些表达式表示 1？

```
-2 + 3, 3 + -2, 3 + (-2), subtract 2 3, 2 + subtract 3
```

标准引导库中有一个函数 flip，定义为

```
flip f x y = f y x
```

请用 flip 表示 subtract。

习题 B Haskell 至少有 3 种定义求幂的方法：

```
(^)  :: (Num a, Integral b) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
(**) :: (Floating a) => a -> a -> a
```

运算 (^) 表示任意数的非负整数次方；运算 (^^) 表示任意数的任意整数（包括负整数）次方；运算 (**) 的两个参数都是分数。运算 (^) 的定义基本上使用第 2 章 Dick 的方

法（见第2章习题F）。请问如何定义(^^)？

习题C 能够使用下列方法定义 div 吗？

```
div :: Integral a => a -> a -> a
div x y = floor (x/y)
```

习题D 再考虑 Clever Dick 给 floor 的定义：

```
floor :: Float -> Integer
floor = read . (takeWhile (/= '.') . show
```

59

为什么这个定义不可行？

考虑下列与 GHCi 的微交互：

```
ghci> 12345678.0 :: Float
1.2345678e7
```

Haskell 允许使用所谓的*科学计数法*（scientific notation），也称*幂计数法*（exponent notation）表示某些浮点数。例如，上面的数表示 1.2345678×10^7。当浮点数的位数很多时，Haskell 用这种形式打印数。请给出另一个理由，说明为什么 Clever Dick 的定义不可行。

习题E 函数 isqrt :: Float -> Integer 返回一个（非负）数的平方根的底。仿照3.3节的方法构造 isqrt x，使其运行时间正比于 $\log x$ 步。

习题F Haskell 提供一个函数 sqrt :: Floating a => a -> a，它给出一个（非负）数的平方根的合理近似值。不过，让我们来自己定义这个函数。如果 y 是$\sqrt{x}$的近似值，那么 x/y 也是$\sqrt{x}$的近似值。而且，或者 $y \leqslant \sqrt{x} \leqslant x/y$，或者 $x/y \leqslant \sqrt{x} \leqslant y$。比 y 和 x/y 更好的$\sqrt{x}$的近似值是什么？（是的，你正在重新发现求平方根的牛顿方法。）

剩下唯一的问题是决定近似值什么时候就足够好了。一种可能的测试是 $|y^2 - x| < \varepsilon$，其中 $|x|$ 表示 x 的绝对值，ε 是一个适当小的数。这个测试保证绝对误差不超过 ε。另一种测试是 $|y^2 - x| < \varepsilon x$，保证相对误差不超过 ε。假定类型 Float 的数只精确到6位有效数字，请问两种测试中哪一种更合理？ε 的合理值是多少？

请由此给出 sqrt 的定义。

习题G 请给出 Nat 作为类族 Ord 实例的显式定义。由此给出下列方法的一个定义：

60

```
divMod :: Nat -> Nat -> (Nat,Nat)
```

3.6 答案

习题A答案 除3 + -2和2 + subtract 3之外，而且这两个式子都不是合式表达式。定义 subtract = flip (-)。

习题B答案

```
x ^^ n = if 0 <= n then x^n else 1/(x ^ (negate n))
```

习题C答案 不能。需要写成

```
div :: Integral a => a -> a -> a
div x y = floor (fromIntegral x / fromIntegral y)
```

习题D答案 Clever Dick 的函数给出 floor (-3.1) = -3，而正确答案是 -4。

如果在结果是负数的情况下通过减 1 设法修改他的定义，那么 `floor (-3.0) = -4`，但是正确答案是 -3。也不行！

另外，Clever Dick 的解给出 `floor 12345678.0 = 1`，因为参数显示为 1.2345678e7。

习题 E 答案

```
isqrt :: Float -> Integer
isqrt x = fst (until unit (shrink x) (bound x))
          where unit (m,n) = (m+1 == n)

shrink :: Float -> Interval -> Interval
shrink x (m,n) = if (p*p) `leq` x then (p,n) else (m,p)
                 where p = (m+n) `div` 2

bound :: Float -> Interval
bound x = (0,until above (*2) 1)
          where above n = x `lt` (n*n)
```

函数`leq`和`lt`在 3.3 节有定义。注意表达式`(p*p) `leq` x`和`x `lt` (n*n)`中的括号。这里没有说明`leq`和`lt`的结合次序，所以如果没有括号的话，两个表达式会被解释成病态的表达式`p *(p `leq` x)`和`(x `lt` n) *n`。（实际上我第一次输入本答案时犯了这个错误。）

61

习题 F 答案 比 y 和 x/y 更好的$\sqrt{x}$的近似值是 $(y+x/y)/2$。相对误差更合理，而且程序如下：

```
sqrt :: Float -> Float
sqrt x = until goodenough improve x
         where goodenough y = abs (y*y-x) < eps*x
               improve y    = (y+x/y)/2
               eps          = 0.000001
```

习题 G 答案 只要定义(<)即可：

```
instance Ord Nat where
  Zero < Zero     = False
  Zero < Succ n   = True
  Succ m < Zero   = False
  Succ m < Succ n = (m < n)
```

现在可以定义：

```
divMod :: Nat -> Nat -> (Nat,Nat)
divMod x y = if x < y then (Zero,x)
             else (Succ q,r)
             where (q,r) = divMod (x-y) y
```

3.7 注记

关于计算机算术的最基本参考资料是 Don Knuth 的《The Art of Computer Programming, Volume 2: Semi-numerical Algorithms》(Addison-Wesley, 1998)。取底和其他简单数值函数的深入研究参见 Don Knuth、Ronald Graham 和 Oren Patashnik 所著的《Concrete Mathematics》(Addison-Wesley, 1989)。

62

第4章
Thinking Functionally with Haskell

列　表

列表是函数式程序设计的引擎。函数之间的数据获取和传递可用列表完成。列表可以拆分、重组或者与其他列表结合形成新的列表。数的列表可以求和或者求积，字符列表可以被读写，等等。总之，列表上的有用运算可以列出很多。本章描述一些常用的列表运算，不过，有一类特别重要的运算将在第6章讨论。

4.1 列表记法

如前所讲，类型[a]表示元素为a的列表。空列表用[]表示。用户可以构造任何类型的列表，但是在同一个列表中不可以包含不同类型的元素。例如：

```
[undefined,undefined] :: [a]
[sin,cos,tan]         :: Floating a => [a -> a]
[[1,2,3],[4,5]]       :: Num a => [[a]]
["tea","for",2]          not valid
```

列表记法，如[1,2,3]实际上是下面更基本记法的简写：

```
1:2:3:[]
```

运算(:) ::a -> [a] -> [a]，读作“cons”，是列表的构造函数。因为该运算是右结合的，所以上面的表达式无需圆括号。该运算没有相关联的定义，所以称为构造函数。换言之，不存在化简诸如1:2:[]的表达式的规则。运算(:)对于两个参数都是非严格的，更确切地说，它是非严格的，并返回非严格的函数。表达式


```
undefined : undefined
```

可能不是很有趣，但是，可以确定的是，它不是空列表。事实上，这也是我们对该列表知道的唯一信息。注意，表达式中两个undefined具有不同的类型。

空列表[]也是列表的一个构造函数。在Haskell中作为数据类型的列表具有下列说明：

```
data List a = Nil | Cons a (List a)
```

唯一的区别是List a写成了[a]，Nil写成了[]，Cons写成了(:)。

根据以上说明，类型为[a]的每个列表具有下列三种形式之一：

- 无定义的列表undefined ::[a]；
- 空列表[] ::[a]；
- 形如x:xs的列表，其中x ::a，xs ::[a]。

由此得出列表有三种：

- 利用(:)和[]构造的有穷（finite）列表，如1:2:3:[]。
- 利用(:)和undefined构造的非完整（partial）列表或者部分列表，如列表filter (<4) [1..]是非完整列表1:2:3:undefined。我们知道在3之后不存在小于4的整数，但是Haskell只是一个计算器，不是定理证明器，所以Haskell继续一直向后检查，而且没有找到满足条件的数字。

- 只用(:)构造的无穷（infinite）列表，如[1..]是非负整数的无穷列表。

三种列表都会出现在日常程序设计中。第9章将探讨无穷列表及其应用。例如，预定义函数 iterate 返回一个无穷列表：

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x:iterate f (f x)
```

特别是，iterate (+1) 1 是正整数的无穷列表，该列表也可写成[1..]（参见4.2节）。

另一个例子是

64

```
head (filter perfect [1..])
 where perfect n = (n == sum (divisors n))
```

它返回第一个完全数，即6，只是目前没人知道 filter perfect [1..]是一个无穷列表还是非完整列表。

最后，可以定义：

```
until p f = head . filter p . iterate f
```

在第3章中函数 until 用于计算取底函数。这个例子也说明，在程序设计中看似简单的函数通常可以用更简单函数的复合表示。这有些像质子和夸克。

4.2　枚举

Haskell 提供枚举整数列表的有用记法。如果 m 和 n 是整数，并且 $m<n$，则可以用下列记法：

[m..n]表示列表 $[m, m+1, \cdots, n]$

[m..]表示无穷列表 $[m, m+1, m+2, \cdots]$

[m,n..p]表示列表 $[m, m+(n-m), m+2(n-m), \cdots, m+a(n-m)]$，其中 a 是满足 $m+a(n-m) \leqslant p$ 的最大整数。

[m,n..]表示无穷列表 $[m, m+(n-m), m+2(n-m), \cdots]$

前两种记法在实际应用中经常出现，后两种出现不太多。例如：

```
ghci> [0,2..11]
[0,2,4,6,8,10]
ghci> [1,3..]
[1,3,5,7,9,11 {Interrupted}
```

第一个例子列表止于10，这是因为11不是偶数。在第二个例子中，很快中断了无穷列表的计算。

事实上，枚举不限于整数，而是类族 Enum 的任何类型都可以使用枚举。这里不再赘述类族 Enum，只说明类型 Char 是 Enum 的成员：

```
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

65

4.3　列表概括

Haskell 提供了另外一种非常有用而且特具魅力的列表记法，称为*列表概括*（list comprehension），可用其他列表构造列表。下面列举几个例子：

```
ghci> [x*x | x <- [1..5]]
[1,4,9,16,25]
ghci> [x*x | x <- [1..5], isPrime x]
[4,9,25]
ghci> [(i,j) | i <- [1..5], even i, j <- [i..5]]
[(2,2),(2,3),(2,4),(2,5),(4,4),(4,5)]
ghci> [x | xs <- [[(3,4)],[(5,4),(3,2)]], (3,x) <- xs]
[4,2]
```

再举一例。假设要生成某个范围的所有毕达哥拉斯三元数，即对于某个给定的 n，使得 $x^2+y^2=z^2$ 而且 $1\leqslant x, y, z\leqslant n$ 的三元组 (x, y, z)。可以定义：

```
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z) | x <- [1..n], y <- [1..n],
                      z <- [1..n], x*x+y*y==z*z]
```

因此

```
ghci> triads 15
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),
 (8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

或许这并非我们想要的结果：每个本质上不一样的三元组用两种不同的方法生成。而且，列表中包括基本三元组的冗余。

为了改进 triad 的定义，可以限制 x 和 y 使得 $x<y$，而且 x 和 y 互素，即它们没有公共因子。数学家知道 $2x^2$ 不可能是一个数的平方，所以第一个限制是正确的。一个数的因子可以如下计算：

```
divisors x = [d | d <- [2..x-1], x `mod` d == 0]
```

因此有

```
coprime x y = disjoint (divisors x) (divisors y)
```

66 这里 disjoint 的定义留作练习。

根据以上讨论，现在可以定义：

```
triads n = [(x,y,z) | x <- [1..n], y <- [x+1..n],
                      coprime x y,
                      z <- [y+1..n], x*x+y*y==z*z]
```

这个定义好于前一个定义，不过还可以尝试让函数运行更快一点，主要目的是说明一个思想。由 $2x^2<x^2+y^2=z^2\leqslant n^2$ 可以推出 $x<n/\sqrt{2}$，所以 $x\leqslant\lfloor n/\sqrt{2}\rfloor$。因此，可以定义：

```
triads n = [(x,y,z) | x <- [1..m], y <- [x+1..n],
                      coprime x y,
                      z <- [y+1..n], x*x+y*y==z*z]
           where m = floor (n / sqrt 2)
```

但是，m 的表达式是不正确的：n 的类型是 Int，整数不能做除法。这里需要一个显式的转换函数，应该使用函数 fromIntegral（注意不是 fromInteger，因为 n 是 Int，而不是 Integer）。需要将 m 定义为 m = floor (fromIntegral n / sqrt 2)。再次强调，必须注意所处理的数的类型，并且了解不同类型数之间的转换函数。

列表概括可用于定义列表上的某些常用函数。例如：

```
map f xs    = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
concat xss  = [x | xs <- xss, x <- xs]
```

实际上，在 Haskell 中情况正好相反：列表概括被翻译成使用 map 和 concat 的等价定义。翻译规则如下：

```
[e |True]        = [e]
[e | q]          = [e | q, True]
[e | b, Q]       = if b then [e | Q] else []
[e | p <- xs, Q] = let ok p = [e | Q]
                       ok _ = []
                   in concat (map ok xs)
```

第四条规则中 ok 的定义使用了不关心（don't care）模式（或者不介意模式），也称为通配符。第四条规则中的 p 是一个模式，并且 ok 的定义说明，对于任何不满足模式 p 的参数，其返回值是空列表。

另一个有用的规则是

67

```
[e | Q1, Q2] = concat [[e | Q2] | Q1]
```

4.4　一些基本运算

列表函数可以通过模式匹配定义。例如：

```
null :: [a] -> Bool
null []     = True
null (x:xs) = False
```

模式[]和 x:xs 不同，而且它们涵盖了所有可能情况，所以，以上 null 定义的两个等式以任何顺序书写都可以。函数 null 是严格的，因为 Haskell 必须通过对参数求值，至少算到可以判断参数是否为空列表，才能决定使用定义的哪个等式。（一个问题是为什么不使用简单的定义 null = (==[])?）以上定义也可以写成

```
null [] = True
null _  = False
```

这个定义使用了不关心模式。

下面是使用模式匹配的另外两个定义：

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

以上定义中不包含模式为[]的等式，所以，如果对 head []或者 tail []求值，则 Haskell 报错。

在模式中也可以用[x]表示 x:[]：

```
last :: [a] -> a
last [x]      = x
last (x:y:ys) = last (y:ys)
```

第一个等式中的模式匹配单元素列表，第二个等式中的模式匹配至少有两个元素的列表。

标准引导函数库的 last 定义稍有区别：

```
last [x]    = x
last (_:xs) = last xs
```

68

定义中使用了不关心模式。两个等式必须按照这种顺序书写，因为 x:[] 与两个模式均匹配。

4.5 串联

下面是串联运算（++）的定义：

```
(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

定义在第一个参数上，而不是第二个参数上使用了模式匹配。第二个等式很简洁，需要一点思考来理解，但是，一旦理解后，读者会对函数程序中列表如何运作豁然大悟。下面是一个简单的求值过程：

```
  [1,2] ++ [3,4,5]
=  {表示法}
  (1:(2:[])) ++ (3:(4:(5:[])))
=  {++的第二个等式}
  1:((2:[]) ++ (3:(4:(5:[]))))
=  {同前}
  1:(2:([] ++ (3:(4:(5:[])))))
=  {++的第一个等式}
  1:(2:(3:(4:(5:[]))))
=  {表示法}
  [1,2,3,4,5]
```

如该例所示，对 xs ++ys 求值的代价与 xs 的长度成正比，其中长度定义为

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

同时注意下面的列表：

```
undefined ++ [1,2] = undefined
[1,2] ++ undefined = 1:2:undefined
```

69

对第一个列表一无所知，但是，知道第二个列表的开始元素是1，下一个元素是2。

串联是满足结合律的运算，所以对于任意列表 xs、ys 和 zs 下列等式成立：

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

第6章将讨论如何证明这样的断言。

4.6 函数 cancat、map 和 filter

读者已经看到列表上的3个最有用的运算：concat、map 和 filter。下面是这些函数利用模式匹配的定义：

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x:map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs
                  else filter p xs
```

这些定义中包含一个共同的主题，将会在第6章进一步探讨。函数 filter 的另一种定义是

```
filter p = concat . map (test p)
test p x = if p x then [x] else []
```

在这个定义中，filter p 首先将列表中满足条件 p 的每个元素转换为单元素列表，不满足条件 p 的转换为空列表，然后将这些列表串联。

函数 map 的两个基本性质是

```
map id      = id
map (f . g) = map f . map g
```

70

第一个等式表示将恒等函数应用于列表的每个元素，结果与原列表相同。这个定律中等式两边的 id 有不同的类型：左边的 id 类型是 a -> a，而右边的 id 类型是[a] -> [a]。第二个等式表示将 g 应用于一个列表的每个元素，然后将 f 应用于前面结果的每个元素，最后结果等同于将 f . g 应用于原列表的每个元素。如果将第二个等式从右向左读，该等式表示对一个列表的两次遍历可以用一次遍历代替，其好处是效率更高。

以上两个等式称为 map 的*函子*（functor）定律。这个术语来自一个数学分支——范畴论。实际上，Haskell 提供了一个类族 Functor，其定义如下：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

方法 fmap 应该与 map 一样满足同样的定律。定义这个类族的原因是，将函数应用于列表的思想可以推广到将函数应用于任意数据结构，如各种树形结构。例如，考虑下列叶结点带标记的二叉树类型：

```
data Tree a = Tip a | Fork (Tree a) (Tree a)
```

具有树状结构的数据会在许多场合出现，如各种表达式的语法具有树状结构。在树上也可以定义函数的映射，不叫 mapTree，而是只要将树定义为类族 Functor 的成员，就可以称之为 fmap：

```
instance Functor Tree where
  fmap f (Tip x)    = Tip (f x)
  fmap f (Fork u v) = Fork (fmap f u) (fmap f v)
```

事实上，列表作为类族 Functor 实例，map 只是 fmap 的同义词：

```
ghci> fmap (+1) [2,3,4]
[3,4,5]
```

前面曾讲过，定义类族 Functor 的主要目的是实现可以映射到列表上的函数也可以映射到其他数据结构上，Haskell 已经实现了这一点，并且引入了一个合适的类族。在今后的章节，特别是第 12 章，读者将看到 map 的函子定律出现在许多计算中。

71 函数 map 还满足另外一组定律，并且具有相同的主题。考虑下面的等式：

```
f . head       = head . map f
map f . tail   = tail . map f
map f . concat = concat . map (map f)
```

第一个等式仅当 f 是严格函数时成立，但是另外两个等式对于任何函数 f 都成立。如果将第一个等式两边均应用于空列表，则有

```
f (head []) = head (map f []) = head []
```

因为空列表的首元素是无定义的，故只有 f 是严格函数时该等式才成立。

以上每个定律都有简单的解释。对于每种情况，可以先把运算（head、tail 等）应用于一个列表，然后改变每个元素，或者先改变每个元素，然后应用这些运算。这些共同点源于这些运算的类型是

```
head   :: [a] -> a
tail   :: [a] -> [a]
concat :: [[a]] -> [a]
```

这些运算的关键点是不依赖于列表的特性，它们只是在列表上移动、舍弃或者提取元素的简单函数。所以它们的类型是多态的。而且，具有多态类型的函数均满足定律：可以先修改值再应用这个函数，也可以先应用函数再修改值。在数学上这种函数称为*自然变换*（natural transformation），相关的定律称为*自然律*（natural law）。

再看一个例子。因为 reverse :: [a] -> [a]，所以期望下列等式成立：

```
map f . reverse =  reverse . map f
```

情况确实如此。当然，该自然律还有待证明。

另一个定律是

```
concat . map concat = concat . concat
```

这个定律表示，等式两边串联由列表的列表构成的列表的两种方式的结果是一样的（或者先进行内部串联，或者先做外部的串联）。

最后是 filter 的一个性质：

72

```
filter p . map f = map f . filter (p . f)
```

可以用等式推导的方式证明这个定律：

```
  filter p . map f
=  {filter的第二个定义}
  concat . map (test p) . map f
=  {map的函子性质}
  concat . map (test p . f)
=  {因为test p . f = map f . test (p . f)}
  concat . map (map f . test (p . f))
=  {map的函子性质}
```

```
  concat . map (map f) . map (test (p . f))
=  {concat的自然律}
  map f . concat . map (test (p . f))
=  {filter的第二个定义}
  map f . filter (p . f)
```

上面这些定律不仅在学术上有价值，而且可用于发现新的、更好的表达定义的方法。这就是为什么函数式程序设计是有史以来最好的程序设计方法。

4.7　函数 zip 和 zipWith

最后，为了给出一个完整简单的常用运算的工具箱，再来考虑函数 zip 和 zipWith。它们在标准引导库中的定义如下：

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _      _      = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

细心的程序员（不喜欢“不关心”模式的程序员）可能给出如下定义：

```
zip [] ys         = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

73

两个定义均使用在两个参数上的模式匹配。必须明白的是，模式匹配的次序是从上到下，自左到右。因此，根据定义有

```
zip [] undefined = []
zip undefined [] = undefined
```

函数 zip 的定义可以用另一种方式表达：

```
zip = zipWith (,)
```

这里的运算(,)是二元组的一个构造函数：(,) a b = (a,b)。

下面是使用 zipWith 的一个例子。假定要检查一个列表元素是否呈非递减序。一种直接定义方法可能是

```
nondec :: (Ord a) => [a] -> Bool
nondec []       = True
nondec [x]      = True
nondec (x:y:xs) = (x <= y) && nondec (y:xs)
```

但是，另一种等价而且更简短的定义是

```
nondec xs = and (zipWith (<=) xs (tail xs))
```

函数 and 是另一个常用的标准引导库函数。如果一个布尔值列表的所有元素均为 True，则该函数返回 True，否则返回 False：

```
and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

再来看最后一个例子。考虑定义一个函数 position，对于一个值 x 和一个有穷列表 xs，该函数返回 x 在列表中出现的第一个位置（位置从 0 开始计算）。如果 x 在 xs 中不出现，则返回 -1。该函数可以如下定义：

```
position :: (Eq a) => a -> [a] -> Int
position x xs
  = head ([j | (j,y) <- zip [0..] xs, y==x] ++ [-1])
```

表达式 zip [0..] xs 将 xs 的每个元素与其在 xs 中的位置配对。尽管第一个参数是无穷列表，但是，如果 xs 是有穷的，那么结果也是有穷列表。注意以上的解决方法是，
74 首先计算 x 出现的所有位置列表，然后取第一个元素。根据惰性计算策略，计算一个列表的第一个元素不必构造列表的所有元素，所以，这种解法在效率上并无损失。可以看出，这种用所有的查找结果来表达一个查找结果是多么简单！

4.8 高频词的完整解

现在给出 1.3 节函数 commonWords 的完整定义。已经定义了：

```
commonWords :: Int -> [Char] -> [Char]
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

需要进一步给出定义的是下列函数：

```
showRun   countRuns   sortRuns   sortWords
```

其他函数，包括 words，是标准 Haskell 库函数。

第一个函数定义简单：

```
showRun :: (Int,Word) -> [Char]
showRun (n,w) = w ++ ": " ++ show n ++ "\n"
```

第二个函数可以如下定义：

```
countRuns :: [Word] -> [(Int,Word)]
countRuns []     = []
countRuns (w:ws) = (1+length us,w):countRuns vs
                   where (us,vs) = span (==w) ws
```

其中引导库函数 span p 将列表拆分为两个列表，第一个是列表中的最大前缀，其所有元素都满足性质 p，第二个列表是剩余的后缀。下面是定义：

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span p []     = ([],[])
span p (x:xs) = if p x then (x:ys,zs)
                else ([],x:xs)
                where (ys,zs) = span p xs
```

现在余下 sortRuns 和 sortWords 尚未定义。可以使用下列命令从模块 Data.
75 List 输入函数 sort：

```
import Data.List (sort)
```

因为 sort 的类型为 sort :: (Ord a) =>[a] ->[a]，所以定义：

```
sortWords :: [Word] -> [Word]
sortWords = sort

sortRuns :: [(Int,Word)] -> [(Int,Word)]
sortRuns = reverse . sort
```

理解第二个定义需要明白 Haskell 在二元组上自动定义了比较运算（<=）：

```
(x1,y1) <= (x2,y2) = (x1 < x2) || (x1 == x2 && y1 <= y2)
```

此外，还需要明白 sort 将元素按照递增序排列。因为需要按照词频数递减序排列，所以先按照词频数递增序排列，然后对列表取逆序。顺便说明，这也是为什么把词频数放在词前，而不是放在词后的原因。

用户也可以定义自己的排序函数，代替使用库排序函数。一种好方法是使用分治（divide and conquer）策略：如果列表最多有一个元素，那么它已经有序；否则将列表分为等长的两个列表，分别递归地对每个小列表排序，然后把两个有序列表合并。由此给出下列定义：

```
sort :: (Ord a) => [a] -> [a]
sort []  = []
sort [x] = [x]
sort xs  = merge (sort ys) (sort zs)
           where (ys,zs) = halve xs

halve xs = (take n xs, drop n xs)
           where n = length xs `div` 2
```

剩下的任务是定义 merge，将两个有序列表归并为一个有序列表：

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys
```

76

实际上，许多 Haskell 程序员不会完全这样书写 merge 定义的最后一个子句。他们会如下书写：

```
merge xs'@(x:xs) ys'@(y:ys)
  | x <= y    = x:merge xs ys'
  | otherwise = y:merge xs' ys
```

这个定义使用了等同模式（as-pattern）。可以看出其意义：代之以将列表分解，然后再重构（廉价但并非无代价的运算）的更好方法是，重用匹配的值。的确如此，不过这也使得简单的数学等式变得有些模糊，在本书将非常保守地使用这种模式。

函数 sort 和 merge 都是递归定义的，值得说明的是，这两个递归均会终止。对于 merge 来说，每次递归调用的两个参数中必有一个参数在变小。因此，有穷步后必能划归到其中的一个递归基。对于 sort 来说，必须注意到如果 xs 的长度至少为 2，那么 ys 和 zs 的长度均严格地小于 xs 的长度，所以同样会终止。但是，如果省略子句 sort [x] = [x]会出现什么情况。因为 1 div 2 =0，故有

```
sort [x] = merge (sort []) (sort [x])
```

这表示 sort [x]的求值需要对 sort [x]求值，所以，对于非空列表情况，sort 的整个定义陷入一个无穷循环之中。定义递归函数时，其中最重要的一部分是检查定义已包含了所有必需的基本情况。

4.9 习题

习题 A 下列等式哪些对所有的 xs 成立，哪些不成立？

```
[]:xs = xs
[]:xs = [[],xs]
xs:[] = xs
xs:[] = [xs]
xs:xs = [xs,xs]
[[]] ++ xs = xs
[[]] ++ xs = [[],xs]
[[]] ++ [xs] = [[],xs]
[xs] ++ [] = [xs]
```

77

另外，为什么没有定义 null = (==[])？

习题 B 假设要列出自然数的所有不同数对 (x, y)。列出数对的顺序不重要，只要能够列出所有数对即可。请问下面的定义是否可行？

```
allPairs = [(x,y) | x <- [0..], y <- [0..]]
```

如果认为该定义不能完成这项任务，请给出一种定义。

习题 C 定义下列函数：

```
disjoint :: (Ord a) => [a] -> [a] -> Bool
```

该函数检查两个递减序列表是否有公共元素。

习题 D 在什么条件下，以下定义给出相同的结果？

```
[e | x <- xs, p x, y <- ys]
[e | x <- xs, y <- ys, p x]
```

比较对两个表达式求值的代价。

习题 E 当伟大的印度数学家 Srinivasan Ramanujan 因病在伦敦住院时，英国数学家 G. H. Hardy 去探望他。Hardy 试图找个话题，就说他是乘出租车来的，车号是 1729，一个对他来说很平凡的数字。Ramanujan 立即回答说，不然，这是第一个能够用两种本质不同的方式表示成两个数的立方和的数：$1^3 + 12^3 = 9^3 + 10^3 = 1729$。请写一个程序，并计算出第二个这样的数。

事实上，定义一个返回范围在 $0 < a, b, c, d \leqslant n$，满足 $a^3 + b^3 = c^3 + d^3$ 的所有本质上不同的四元组 (a, b, c, d) 的函数，建议使用列表概括，但是必须仔细想清楚两个四元组本质不同的含义。总之，$a^3 + b^3 = c^3 + d^3$ 有 8 种不同的写法。

78

习题 F 构造列表的对偶方法是在列表的尾部添加元素：

```
data List a = Nil | Snoc (List a) a
```

当然，Snoc 是 Cons 的反写。利用这种方式，列表 [1, 2, 3] 可表示成

```
Snoc (Snoc (Snoc Nil 1) 2) 3
```

两种不同的观点提供了完全相同的信息，但是组织方式不同。请给出 Snoc 观点的函数 head 和 last 的定义，并定义下列两个在两种观点之间转换的函数：

```
toList :: [a] -> List a
fromList :: List a -> [a]
```

（提示：reverse 是有效的，翻转一个列表是线性时间。）

习题 G　对 length xs 求值需要多长时间？考虑 length 的另一种定义：

```
length :: [a] -> Int
length xs = loop (0,xs)
  where loop (n,[])   = n
        loop (n,x:xs) = loop (n+1,xs)
```

需要的空间有变化吗？如果转用勤奋求值空间有变化吗？这些问题将在第 7 章详细讨论。

习题 H　引导库函数 take n 取得一个列表的前 n 个元素，函数 drop n 舍弃前 n 个元素。请给出这些函数的递归定义。根据给出的定义，下列表达式的值是什么？

```
take 0 undefined     take undefined []
```

一个更难的问题是，能否给出一个定义，使得以上表达式的值均为[]？如果不能，为什么？

79

下列等式中哪些对于所有整数 m 和 n 都成立？不必说明理由，只要设法理解表达式的含义即可。

```
take n xs ++ drop n xs  =  xs
take m . drop n  =  drop n . take (m+n)
take m . take n  =  take (m `min` n)
drop m . drop n  =  drop (m+n)
```

标准引导库函数 splitAt n 可以如下定义：

```
splitAt n xs = (take n xs,drop n xs)
```

不过，显然以上定义效率比较低，因为这里需要遍历列表 xs 两次。请给出只需要遍历 xs 一次的 splitAt 的定义。

习题 I　对于等式：

```
map (f . g) xs = map f (map g xs)
```

下列命题中，同意哪些，不同意哪些（同意无需给出理由）？

1. 等式并非对所有 xs 成立，命题的真假依赖于 xs 是否是有穷列表。

2. 等式并非对所有 f 和 g 成立，命题的真假依赖于 f 和 g 是否是严格函数。

3. 等式对于所有列表，包括有穷、非完整和无穷列表，以及对于所有合适类型的函数 f 和 g 都成立。事实上，map (f . g) = map f . map g 是更简洁的表达方式。

4. 等式看似正确，但是需要根据 map 和函数复合的定义证明。

5. 从左至右来看等式，它表达了一种程序优化：对一个列表的两次遍历可以用一次遍历完成。

6. 在惰性求值策略下并不是优化，因为 map f 求值没有开始前，map g xs 不会被完全计算。

7. 等式右边无论是分部计算还是整体计算，中间都会产生一个中间列表，但是等式

80 左边则不会。所有，即使在惰性求值策略下，该等式也表示一种程序优化。

习题 J 下面列出的等式中至少有一个是错误的。请识别哪些成立，哪些不成立。同样，无需说明理由，目的是理解等式的含义。

```
map f . take n    = take n . map f
map f . reverse   = reverse . map f
map f . sort      = sort . map f
map f . filter p  = map fst . filter snd . map (fork (f,p))
filter (p . g)    = map (invertg) . filter p . map g
reverse . concat  = concat . reverse . map reverse
filter p . concat = concat . map (filter p)
```

在第五个等式中假定 invertg 满足 invertg . g = id。第四个等式中的函数 fork 定义如下：

```
fork :: (a -> b,a -> c) -> a -> (b,c)
fork (f,g) x = (f x, g x)
```

习题 K 定义函数 unzip 和 cross 如下：

```
unzip = fork (map fst, map snd)
cross (f,g) = fork (f . fst, g . snd)
```

这些函数的类型是什么？

利用简单的等式推理证明下列等式：

```
cross (map f, map g) . unzip = unzip . map (cross (f,g))
```

证明可以使用 map 的函子律和下列规则：

```
cross (f,g) . fork (h,k) = fork (f . h,g . k)
fork (f,g) . h           = fork (f . h,g . h)
fst . cross (f,g)        = f . fst
snd . cross (f,g)        = g . snd
```

习题 L 接着上一个习题，证明

81

```
cross (f,g) . cross (h,k) = cross (f . h,g . k)
```

还有 cross (id, id) = id（为什么?）。所以，cross 除了参数是一个二元组外，看似具有函子的性质。是的，cross 是一个双函子（bifunctor）。这也提示可以给出下列推广：

```
class Bifunctor p where
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
```

其中 bimap 的参数是并列的，而不是一个二元组。请用类型 Pair 的 Bifunctor 实例 bimap 表示 cross，其中：

```
type Pair a b = (a,b)
```

现在考虑数据类型：

```
data Either a b = Left a | Right b
```

请将 Either 作为 Bifunctor 的实例，给出实例定义。

4.10 答案

习题 A 答案　只有下列 3 个等式成立：

```
xs:[] = [xs]
[[]] ++ [xs] = [[],xs]
[xs] ++ [] = [xs]
```

如果定义 `null = (==[])`，那么它的类型会有限制：

```
null :: (Eq a) => [a] -> Bool
```

这意味着，只有列表中元素类型可比较相等时，列表才可以比较相等。当然，空列表不含任何元素，所以不需要 `(==)`。

习题 B 答案　答案是否定的，`allPairs` 生成下列无穷列表：

```
allPairs = [(0,y) | y <- [0..]]
```

一种方法是使用下列定义，按照二元组之和递增的顺序列出所有二元组。

```
allPairs = [(x,d-x) | d <- [0..], x <- [0..d]]
```

82

习题 C 答案　定义如下：

```
disjoint xs [] = True
disjoint [] ys = True
disjoint xs'@(x:xs) ys'@(y:ys)
  | x < y  = disjoint xs ys'
  | x == y = False
  | x > y  = disjoint xs' ys
```

为了使得定义更聪明，这里使用了等同模式。

习题 D 答案　只有 `ys` 是有穷列表时，它们的结果才相同：

```
ghci> [1 | x <- [1,3], even x, y <- undefined]
[]
ghci> [1 | x <- [1,3], y <- undefined, even x]
*** Exception: Prelude.undefined
ghci> [1 | x <- [1,3], even x, y <- [1..]]
[]
Prelude> [1 | x <- [1,3], y <- [1..], even x]
{Interrupted}
```

当它们结果相同时，前者更高效。

习题 E 答案　一种生成本质不同的四元组方法是限制四元组 (a, b, c, d) 满足条件 $a \leqslant b$、$c \leqslant d$ 以及 $a < c$。因此

```
quads n = [(a,b,c,d) | a <- [1..n],  b <- [a..n],
                       c <- [a+1..n],d <- [c..n],
                       a^3 + b^3 == c^3 + d^3]
```

第二个这样的数是 $4104 = 2^3 + 16^3 = 9^3 + 15^3$。

习题 F 答案

```
head :: List a -> a
head (Snoc Nil x) = x
head (Snoc xs x)  = head xs
```

83

```
last :: List a -> a
last (Snoc xs x) = x

toList :: [a] -> List a
toList = convert . reverse
   where convert []     = Nil
         convert (x:xs) = Snoc (convert xs) x
fromList :: List a -> [a]
fromList = reverse . convert
   where convert Nil         = []
         convert (Snoc xs x) = x:convert xs
```

习题 G 答案 因为在内存中需要构建下列表达式，故所需的空间是线性的。

```
1 + (1 + (1 + ... (1 + 0)))
```

在惰性求值策略下，length 的第二个定义的空间需求不变，因为在内存中需要构建下列表达式：

```
loop ((((0 + 1) + 1) + 1 ... +1),[])
```

但是在勤奋求值策略下，一个列表的长度可以使用额外的常数空间计算出来。

习题 H 答案

```
take, drop :: Int -> [a] -> [a]
take n []     = []
take n (x:xs) = if n==0 then [] else x:take (n-1) xs

drop n []     = []
drop n (x:xs) = if n==0 then x:xs else drop (n-1) xs
```

按照 take 的这个定义则有

```
take undefined [] = []   take 0 undefined = undefined
```

利用 take 的另外定义：

```
take n xs | n==0      = []
          | null xs   = []
          | otherwise = head xs: take (n-1) (tail xs)
```

84 则有

```
take undefined [] = undefined   take 0 undefined = []
```

对这个复杂问题的答案是否定的。参数 n 或者参数 xs 必须被检查，无论哪个在先，如果该参数的结果是⊥，则最后结果是⊥。

无论哪个定义，所有 4 个等式对于所有列表 xs 和所有 m、n 都成立，其中 m、$n \neq \perp$。

函数 splitAt n 可以定义如下：

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n [] = ([],[])
splitAt n (x:xs) = if n==0 then ([],x:xs) else (x:ys,zs)
                   where (ys,zs) = splitAt (n-1) xs
```

习题 I 答案 同意3、4、5和7。

习题 J 答案 唯一错误的等式是 map f . sort = sort . map f，该等式只有在 f 保

序的情况下成立，即 $x \leqslant y \equiv fx \leqslant fy$。

习题 K 答案

```
unzip :: [(a,b)] -> ([a],[b])
cross :: (a -> b, c -> d) -> (a,c) -> (b,d)
```

计算过程如下：

```
 cross (map f, map g) . unzip
=  {unzip的定义}
 cross (map f, map g) . fork (map fst, map snd)
=  {cross和fork的定律}
 fork (map f . map fst, map g . map snd)
=  {map的定律}
 fork (map (f . fst), map (g . snd))
```

85

看似卡住了，因为没有规则可用。再尝试计算等式右边：

```
 unzip . map (cross (f,g))
=  {unzip的定义}
 fork (map fst, map snd) . map (cross (f,g))
=  {fork定律}
 fork (map fst . map (cross (f,g)),
       map snd . map (cross (f,g)))
=  {map定律}
 fork (map (fst . cross (f,g)),
       map (snd . cross (f,g)))
=  {fst和snd的定律}
 fork (map (f . fst), map (g . snd))
```

好啊，等式两边都化简到了同一个表达式。这也是常用的计算方法：等式的一边并不总是很容易化简成另一边，但是两边可以化简到同一个式子。

目前看到的计算都是在函数层进行的。这种风格的定义和证明称为*点自由的*（point-free，也有人戏称为*无意义的*（pointless））。第 12 章的自动计算器产生点自由证明。点自由风格非常灵活，但是这种风格必须使用各种*管道套结组合子*（plumbing combinator）为函数传递参数，如 `fork` 和 `cross`。套结组合子可以推送值，重复它们，甚至删除它们。最后一种组合子的例子如：

```
const :: a -> b -> a
const x y = x
```

这个组合子在标准引导库中有定义，有时非常有用。

标准引导库中定义的另外两个套结组合子是 `curry` 和 `uncurry`：

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

一个*卡瑞化*（curried）的函数每次取得一个参数，而*非卡瑞化*（non-curried）的函数一次取得用多元组表示的参数。卡瑞化函数的主要优点是函数可以*部分应用*（partially ap-

86

plied)。例如，take n 本身是完全合理的函数，map f 同样是一个函数。这也是一开始就使用卡瑞化函数的原因。

顺便说明的是，卡瑞化函数是以美国逻辑学家 Haskell B. Curry 命名的。是的，Haskell 也是这样命名的。

习题 L 答案

```
  cross (f,g) . cross (h,k)
=   {definition of cross}
  cross (f,g) . fork (h . fst, k . snd)
=   {law of cross and fork}
  fork (f . h . fst,g . k . snd)
=   {definition of cross}
  cross (f . h, g . k)
```

另有 cross = uncurry bimap，其中 uncurry 如习题 K 答案中定义。

下面是 Either 的实例定义：

```
instance Bifunctor Either where
  bimap f g (Left x)  = Left (f x)
  bimap f g (Right y) = Right (g y)
```

4.11 注记

本章介绍的大多数函数都可以在 Haskell 标准引导库中找到。函子、双函子以及自然变换的解释可以在有关范畴论书中找到，这里列出两本：Benjamin Pierce 编著的《Basic Category Theory for Computer Scientists》(MIT Press，1991)、Richard Bird 和 Oege de Moor 编著的《The Algebra of Programming》(Prentice Hall，1997)。

关于定律内容，请参见 Phil Wadler 的著名文章“Theorem for free!”，可在下列链接找到：

homepages. inf. ed. ac. uk/wadler/papers/free/

87
~
88

数学上的所谓的士数 taxicab(n) 是能够用 n 种方法表示成两个正整数立方和的最小数，如 1729 是 taxicab(2)。关于士数的更多信息请搜索“taxicab numbers”。

一个简单的数独求解器

数独玩法：在一个9×9的棋盘上填写数字1~9使得每一行、每一列和每个3×3的方格都包含1~9。解决这个谜题无需数学知识，但是需要推理和逻辑知识。

——摘自《独立报》“玩数独的建议”

本章主题是一个扩展练习：使用列表解决问题，使用等式推理对程序推理，并改进其性能。

数独游戏在9×9棋盘上玩，不过其他规格的棋盘也可以玩。给定一个矩阵，如图5-1所示，解法是用数字1~9填充空格，使得每一行、每一列和每个3×3的方格都包含1~9。一般来讲，一个数独可能有许多解，但是一个好的数独谜题应该总是有唯一解。本章的目的是设计一个解数独的程序。特别地，将定义一个函数 `solve` 用于计算完成一个给定谜题的所有可能填充方法的列表。如果只需要一个解，那么只要取列表的第一个元素即可。对于惰性计算，这也意味着只计算出第一个结果。

		4			5	7		
					9	4		
3	6							8
7	2			6				
			4		2			
				8			9	3
4							5	6
		5	3					
		6	1			9		

图5-1 数独棋盘

下面先从问题的说明开始，然后使用等式推理计算更有效的解。这里无需数学，只需要推理和逻辑。

5.1 问题说明

下面是相关的基本数据类型，首先是矩阵：

```
type Matrix a = [Row a]
type Row a    = [a]
```


两个类型同义词仅仅说明，`Matrix a` 是`[[a]]`的同义词。但是，说明的方法强调了一个矩阵是行的列表，更准确地说，一个 $m \times n$ 矩阵是 m 个行的列表，每行均是长度为 n 的列表。Haskell类型同义词不能强加这些约束，但是能够表达这种约束的语言，称为依存类型（dependent-typed）语言。

一个棋盘是一个9×9的数字矩阵：

```
type Grid  = Matrix Digit
type Digit = Char
```

合法的数字是1~9，0表示空白：

```
digits :: [Char]
digits = ['1' .. '9']

blank :: Digit -> Bool
blank = (== '0')
```

注意 Char 也是类族 Enum 的实例，所以['1' .. '9']是合法表达式，且表示非零数字列表。

为简单起见，假定棋盘只包含数字和空白，这样就无需检查棋盘是否有意义。但是，是否需要假定非空白格的数字在每行、每列或者每个方格中不会重复？如果有重复，那么
90 这样的谜题无解。现在暂时抛开这个问题，待看到解决问题的算法如何展开后再做决定。

先看问题说明。目标是写出最简单而且最清晰的说明，先不考虑结果的效率。这是函数式程序设计与其他形式程序设计的关键区别：总是可以从一个清晰而且简单，尽管可能效率极低的 solve 定义开始，然后应用函数式程序设计中的定律修改计算，使其满足时间和空间上的要求。

一种方法是首先构造一个包含所有正确填充棋盘的列表，该列表可能相当长，但依然是有穷的，然后将给定的棋盘与列表中每个棋盘对照，找出与给定棋盘非空白格上数字一致的棋盘。显然，这种方法已将非有效说明用到极致。另一种合理的方法是在给定的棋盘上给每个空白填写所有可能的数字，结果是一个填满的棋盘列表，然后可以在此列表中过滤出那些每行、每列和每块都不含重复数字的棋盘。这种说明的实现方法如下：

```
solve :: Grid -> [Grid]
solve = filter valid . completions
```

其中的辅助函数具有下列类型：

```
completions :: Grid -> [Grid]
valid   :: Grid -> Bool
```

首先考虑 completions，然后考虑 valid。一种定义 completions 的方法是采用两步过程：

```
completions = expand . choices
```

其中：

```
choices :: Grid -> Matrix [Digit]
expand  :: Matrix [Digit] -> [Grid]
```

函数 choices 为每个棋盘位置安放了可能的数字：

```
choices  = map (map choice)
choice d = if blank d then digits else [d]
```

如果一个位置是空白，那么所有的数字都作为可用选择；否则只有一个选择，故返回单元素列表。如果想把函数 f 应用于矩阵的每个元素，需要使用的函数是 map (map f)，因为一个矩阵只是一个列表的列表。

应用 choices 后获得一个矩阵，矩阵的每个元素是一个数字列表。接下来要做的是
91 定义一个函数 expand，通过用所有可能的方式填写棋盘，将该矩阵转换为棋盘列表。这个任务看似难以想象，所以先考虑一个更简单的问题，这里不是 9×9 的矩阵，而是一个长度为 3 的列表。假设要把下列列表：

```
[[1,2,3],[2],[1,3]]
```

转换为列表：

```
[[1,2,1],[1,2,3],[2,2,1],[2,2,3],[3,2,1],[3,2,3]]
```

第二个列表中每个列表产生的方法：用各种可能的方法在第一个列表中取一个元素，在第二个列表中取一个元素，在第三个列表中取一个元素。假设用 cp（笛卡儿积的简写，以上列表也是数学家所称的笛卡儿积）表示这样的函数。似乎看不出使用其他函数计算 cp 的聪明方法，所以仍然使用惯用的方法，即将参数分为两种可能的情况，一种是空列表[]，另一种是非空列表 xs:xss。如果先猜想 cp []的定义，或许会搞错，最好是先考虑第二种非空列表的定义。假设有

```
cp [[2],[1,3]] = [[2,1],[2,3]]
```

如何将这个定义扩充到 cp ([1,2,3]:[[2],[1,3]])？答案是将 1 置于 cp [[2],[1,3]]的每个元素之前，然后将 2 置于该列表的每个元素之前，最后将 3 置于同一个列表的每个元素之前。这个过程可以用列表概括清晰地表达出来：

```
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

换言之，将 xs 的每个元素置于 cp xss 的每个元素之前。

如果读者对低效率有很敏锐的嗅觉，则会发现这样用一行来表达不是最好的方法，事情确实如此。7.3 节将对此进行深入讨论，不过现在可以给出一种更高效的定义：

```
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
              where yss = cp xss
```

这种定义确保 cp xss 只计算一次。

再返回来看，如何定义 cp []呢？答案并非[]，而是[[]]。为了看清楚为什么[]是错误的，考虑下列计算：

```
cp [xs] = cp (xs:[])
        = [x:ys | x <- xs, ys <- cp []]
= [x:ys | x <- xs, ys <- []]
= []
```

92

事实上，如果 cp [] = []，那么可以证明对于所有列表 xss 都有 cp xss = []。所以，第一个定义显然是错误的。读者可以验证第二个答案[[]]确实能给出正确的答案。

综上所述，现在可以将 cp 定义为

```
cp :: [[a]] -> [[a]]
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
              where yss = cp xss
```

例如：

```
ghci> cp [[1],[2],[3]]
[[1,2,3]]

ghci> cp [[1,2],[],[4,5]]
[]
```

对于第二个例子，因为在中间的列表中没有可选元素，所以结果是空列表。

但是如何处理矩阵和矩阵上类似于 cp 的函数 expand 呢？读者需要思考一下，然后才能弄清楚 expand 需要完成的是

```
expand :: Matrix [Digit] -> [Grid]
expand  = cp . map cp
```

这看起来有点神秘，但是 map cp 返回每行所有可能选择的列表。因此，在此结果上应用 cp 之后得到棋盘的所有可能选择。定义右边的一般类型是

```
cp . map cp :: [[[a]]] -> [[[a]]]
```

而 expand 声明的类型只是一般类型的一种限制版本。注意，如果任意一行的任何元素是空列表，那么 expand 返回空列表。

最后，一个合理的棋局是所有行、列或者块都没有重复数字出现：

```
valid :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols g) &&
          all nodups (boxs g)
```

93

引导库函数 all 有如下定义：

```
all p = and . map p
```

将 all p 应用于一个有穷列表 xs 时，如果列表中所有元素满足 p，则结果是 True，否则结果是 False。函数 nodups 可以如下定义：

```
nodups :: (Eq a) => [a] -> Bool
nodups []     = True
nodups (x:xs) = all (/=x) xs && nodups xs
```

将 nodups 应用于长度为 n 的列表所花费的时间与 n^2 成正比。另一种方法是将列表排序，然后检查有序列表严格递增。排序可以在 $n\log n$ 的常数倍时间完成。看起来后一种方法比前一种方法大大节省了时间。但是，对于 $n=9$ 的情况使用有效的排序是否值得，答案并不清楚。问题是，$2n^2$ 步和 $100n\log_2 n$ 步哪个更好？

剩余的任务是定义函数 rows、cols 和 boxs。如果一个矩阵用行的列表表示，那么 rows 只是矩阵上的恒等函数：

```
rows :: Matrix a -> Matrix a
rows = id
```

函数 cols 计算矩阵的转置。所以，如果一个矩阵有 m 行，每行长度为 n，那么转置矩阵有 n 行，每行长度为 m。假设 m 和 n 均不为 0，则可定义：

```
cols :: Matrix a -> Matrix a
cols [xs]     = [[x] | x <- xs]
cols (xs:xss) = zipWith (:) xs (cols xss)
```

通常在矩阵代数中假设矩阵非空，以上定义足以满足这里的应用。但是，有趣的是 m 或者 n 为 0 时会出现什么情况。该问题将在习题中考虑。

函数 boxs 的定义更有趣。下面先给出定义，然后再解释。

```
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup .
       map cols .
       group . map group
```

其中函数 group 将列表以 3 个元素为一组划分为组的列表：

```
group :: [a] -> [[a]]
group [] = []
group xs = take 3 xs:group (drop 3 xs)
```

94

函数 upgroup 将分组的列表转换为不分组列表：

```
ungroup :: [[a]] -> [a]
ungroup = concat
```

在4×4的情况下，group 将列表两两一组划分而不是3个一组时，函数 boxes 的作用如下图所示。

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} ab & cd \\ ef & gh \end{pmatrix} \\ \begin{pmatrix} ij & kl \\ mn & op \end{pmatrix} \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} a & b & e & f \\ c & d & g & h \\ i & j & m & n \\ k & l & o & p \end{pmatrix} \leftarrow \begin{pmatrix} \begin{pmatrix} ab & ef \\ cd & gh \end{pmatrix} \\ \begin{pmatrix} ij & mn \\ kl & op \end{pmatrix} \end{pmatrix}$$

可以看出，分组生成一个矩阵列表，转置其中每个矩阵，然后合并分组生成2×2块，其中矩阵行是原矩阵中的2×2块。

5.2 合法程序的构造

请注意，代之以用下标表示矩阵，通过计算下标确定行、列和块，这里定义的函数均把矩阵本身作为一个整体来处理。这种风格被形象地称为*全麦面粉程序设计*（wholemeal programming）。全麦面粉程序设计对健康有益，它帮助人们避免一种称为过度下标的疾病，并且鼓励合法程序的构造。

例如，下面是数独棋盘上成立的3个定律：

```
rows . rows = id
cols . cols = id
boxs . boxs = id
```

换句话说，3个函数都是对合。前两个等式对于任意矩阵成立，第三个等式对于任意 $n^2 \times n^2$ 阶矩阵成立（只要将 group 的定义改为按照 n 进行分组）。这里有两个等式的证明简单，但是有一个证明比较难。难证的不是读者想象的关于 boxs 的定律，而是关于 cols 的对合性质。尽管直观上对矩阵两次转置得到原矩阵是显然的，但是根据 cols 的定义来证明有点难度，这里不给出证明的细节，主要原因是目前还没有讨论完成这个证明的工具。

95

相比之下，下面是 boxs 的对合性质证明。证明可以通过简单的等式推理完成。证明中使用了各种定律，包括 map 的函子定律，id 是复合的单位元，以及下列事实：

```
ungroup . group = id
group . ungroup = id
```

第二个等式只有在分组列表上成立，但这将是下面计算所满足的条件。

下面将叙述证明过程，不再给出证明的每个细节。出发点是用 boxs 的定义重写 boxs . boxs：

```
map ungroup . ungroup . map cols . group . map group .
map ungroup . ungroup . map cols . group . map group
```

利用 map 的函子律和 group 与 ungroup 互逆的性质，中间的表达式 map group . map ungroup 化简为 id。由此得到

```
map ungroup . ungroup . map cols . group .
ungroup . map cols . group . map group
```

使用 group . ungroup = id 得到

```
map ungroup . ungroup . map cols .
map cols . group . map group
```

由 map 的函子律和 cols 的对合性质得到

```
map ungroup . ungroup . group . map group
```

再利用 ungroup . group = id 两次即可完成证明。可以看出，这是一个很简单的计算。

下面是在 $N^2 \times N^2$ 选择矩阵上成立的另外 3 个定律：

```
map rows . expand = expand . rows
map cols . expand = expand . cols
map boxs . expand = expand . boxs
```

96

稍后会用到这些定律。

最后，下面是关于 cp 的两个定律：

```
map (map f) . cp    = cp . map (map f)
filter (all p) . cp = cp . map (filter p)
```

第一个定律（自然律）是由 cp 的类型蕴涵的，在第 4 章也出现过类似的定律。第二个定律表示，对一个列表的列表做笛卡儿积，然后保留所有元素满足性质 p 的列表，实现这一过程的另一种方法是，首先过滤原列表，只保留满足性质 p 的元素，然后做笛卡儿积。如前面句子所示，一个等式胜过千言万语。

5.3 修剪选择矩阵

总结现在的成果，有

```
solve :: Grid -> [Grid]
solve = filter valid . expand . choices
```

虽然理论上是可运行的，但是 solve 的这个定义是不现实的。假如 81 格中有 20 个已经填充，那么共有 9^{61} 个格子，或者

```
ghci> 9^61
1617309269922988089371861846558644535758328064784065995760 9
```

需要查看这么多格子！所以需要更好的方法。

为了得到更有效的求解器，显然的想法是去除一个格子的那些选择 c，这些选择 c 已经作为单元素填充出现在该格子所在的行、列和块中。一个单元素填充对应于一个已经确

定的选择。因此需要寻找一个函数：

```
prune :: Matrix [Digit] -> Matrix [Digit]
```

使得

```
filter valid . expand = filter valid . expand . prune
```

如何定义函数 prune 呢？因为一个矩阵是行的列表，所以，先对行裁剪是好的开端。函数 pruneRow 定义如下：

```
pruneRow :: Row [Digit] -> Row [Digit]
pruneRow row = map (remove fixed) row
where fixed = [d | [d] <- row]
```

97

确定的选择是每行中的单元素选择。其中 fixed 的定义使用了包含模式的列表概括：row 的元素如果不是单元素模式则被丢弃。

函数 remove 从未确定的选择中删除了已确定的选择：

```
remove :: [Digit] -> [Digit] -> [Digit]
remove ds [x] = [x]
remove ds xs  = filter (`notElem` ds) xs
```

标准引导库函数 notElem 定义如下：

```
notElem :: (Eq a) => a -> [a] -> Bool
notElem x xs = all (/= x) xs
```

下面是函数 pruneRow 应用的几个例子：

```
ghci> pruneRow [[6],[1,2],[3],[1,3,4],[5,6]]
[[6],[1,2],[3],[1,4],[5]]

ghci> pruneRow [[6],[3,6],[3],[1,3,4],[4]]
[[6],[],[3],[1],[4]]
```

在第一个例子中，[6]和[3]是已确定的选择，删除这些选择后使得最后一个格子变成了确定选择。在第二个例子中，删除确定选择使得第二个填充变成空选择列表。

函数 pruneRow 满足等式：

```
filter nodups . cp = filter nodups . cp . pruneRow
```

换句话说，这个等式表示裁剪一行不会舍弃不含重复元素的列表。稍后会用到这个性质。

现在几乎可以进行确定函数 prune 的定义了。几乎是，但不完全是，因为还需要另外两个性质：如果 f . f = id，那么

```
filter (p . f) = map f . filter p . map f
filter (p . f) . map f = map f . filter p
```

98

第二个可由第一个推出（为什么?）。下面是第一个性质的证明：

```
  map f . filter p . map f
=   {第4章证明了
          filter p . map f = map f . filter (p . f)}
  map f . map f . filter (p . f)
=   {map的算子律以及f . f = id}
  filter (p . f)
```

现在进行主要部分计算。出发点是使用 valid 的定义重写表达式 filter . valid . expand:

```
filter valid . expand
 = filter (all nodups . boxs) .
   filter (all nodups . cols) .
   filter (all nodups . rows) . expand
```

过滤器在右边出现的顺序不重要。解决的方案是将每个过滤器送到与 expand 的战斗中。例如，在块的情况中可以计算：

```
 filter (all nodups . boxs) . expand
=  {根据上面filter的定律，因为boxs . boxs = id}
 map boxs . filter (all nodups) . map boxs . expand
=  {因为 map boxs . expand = expand . boxs}
 map boxs . filter (all nodups) . expand . boxs
=  {expand的定义}
 map boxs . filter (all nodups) . cp . map cp . boxs
=  {因为 filter (all p) . cp = cp . map (filter p)}
 map boxs . cp . map (filter nodups) . map cp . boxs
=  {map的函子律}
 map boxs . cp . map (filter nodups . cp) . boxs
```

现在可以使用性质：

```
filter nodups . cp = filter nodups . cp . pruneRow
```

然后重写最后的表达式：

99

```
map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
```

余下的步骤基本上重复以上计算，不过用相反的次序：

```
 map boxs . cp . map (filter nodups . cp . pruneRow) .
 boxs
=  {map的函子律}
 map boxs . cp . map (filter nodups) .
 map (cp . pruneRow) . boxs
=  {因为cp . map (filter p) = filter (all p) . cp}
 map boxs . filter (all nodups) . cp .
 map (cp . pruneRow) . boxs
=  {map的函子律}
 map boxs . filter (all nodups) .
 cp . map cp . map pruneRow . boxs
=  {expand的定义}
 map boxs . filter (all nodups) .
 expand . map pruneRow . boxs
=  {filter的定律，因为boxs . boxs = id}
 filter (all nodups . boxs) . map boxs .
 expand . map pruneRow . boxs
=  {因为map boxs . expand = expand . boxs}
 filter (all nodups . boxs) . expand .
 boxs . map pruneRow . boxs
=  {引入pruneBy f = f . pruneRow . f}
 filter (all nodups . boxs) . expand . pruneBy boxs
```

我们已经证明了：

```
filter (all nodups . boxs) . expand
 = filter (all nodups . boxs) . expand . pruneBy boxs
```

其中 pruneBy f = f . map pruneRow . f。重复对行和列的计算，可以得到

```
filter valid . expand = filter valid . expand . prune
```

其中：

```
prune = pruneBy boxs . pruneBy cols . pruneBy rows
```

100

总结起来，solve 先前的定义可以用以下新定义代替：

```
solve = filter valid . expand . prune . choices
```

事实上，裁剪不仅做一次，而是可以根据需要做很多次。这样做是合理的，因为一轮裁剪后有些选择可能变成了单元素选择，再一轮裁剪又删除了更多不可能的选择。

所以，可以定义：

```
many :: (Eq a) => (a -> a) -> a -> a
many f x = if x == y then x else many f y
           where y = f x
```

然后再次重定义 solve 为

```
solve = filter valid . expand . many prune . choices
```

最简单的数独问题的解决方法是不停地裁剪选择矩阵，直至最后只剩下单元素选择。

5.4 格子的扩展

many prune . choices 的结果是选择的矩阵，可以分成下列三类：

1. 一个完全的矩阵，其中每个元素都是单个选择。在这种情况下，expand 将抽取一个棋盘用于检查其有效性。

2. 一个含元素为空选择列表的矩阵。在这种情况下 expand 将生成空列表。

3. 一个不含空选择列表的矩阵，但是某些元素包含两个以上的选择。

问题是如何处理第三类矩阵。另一种合理方法是不进行完全扩展，而是在仅对矩阵的一个元素做了扩展的矩阵上进行裁剪。希望单格扩展与裁剪的混合能够产生一个更快的解。因此，目的是构造一个只对单格扩展的部分函数：

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
```

101

该函数只对不完全矩阵返回确定的结果，而且在这些矩阵上满足

```
expand = concat . map expand .  expand1
```

事实上这个列表间的等式过强。人们希望保证部分扩展不会失掉可能的选择，但是并不真正关心等式两边生成结果的次序。所以，将以上等式解释为两边在进行置换后相等。

应该在哪个格子上进行扩展呢？简单的方法是从矩阵的第一个非单选择元素开始。设想一个矩阵 rows 分解成如下形式：

```
rows = rows1 ++ [row] ++ rows2
row = row1 ++ [cs] ++ row2
```

其中格子 cs 是 row 中的非单元素选择列表，而 row 又是矩阵 rows 中的一行。

那么可以定义：

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
expand1 rows
 = [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
```

为了将矩阵分解成以上形式，使用引导库函数 break：

```
break :: (a -> Bool) -> [a] -> ([a],[a])
break p = span (not . p)
```

其中函数 span 在 4.8 节定义。例如：

```
ghci> break even [1,3,7,6,2,3,5]
([1,3,7],[6,2,3,5])
```

还需要标准引导库函数 any，其定义为

```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

其中 or 的参数是一个布尔值列表，如果列表中任何元素为 True，则返回 True，否则返回 False：

```
or :: [Bool] -> Bool
or []     = False
or (x:xs) = x || or xs
```

102

最后，测试 single 的定义（使用不关心模式）为

```
single :: [a] -> Bool
single [_] = True
single _   = False
```

现在可以定义：

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
expand1 rows
 = [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
   where
   (rows1,row:rows2) = break (any (not . single)) rows
   (row1,cs:row2)    = break (not . single) row
```

第一个 where 子句将矩阵拆分成两个行的列表，第二个列表的第一个元素是包含非单选择列表的行，第二个 break 再将该行拆分成两个列表，其中第二个列表的第一个元素是第一个非单选择元素。如果矩阵只包含单选择元素，则

```
break (any (not . single)) rows = [rows,[]]
```

而且 expand1 的运行返回一个错误信息。

函数 expand1 的定义的问题是它可能引起无用的工作。如果这样找到的第一个非单选择元素恰好是空列表，那么 expand1 将会返回空列表，但是，假如这个列表在矩阵中藏得很深，那么 expand1 将会做许多无意义的计算，企图找到一个根本不存在的解。合理的、更好的选择扩展格子的方法是找出具有最少选择（当然不是 1）的格子。一个没有选择的格子意味着这个谜题不可解，所以，尽快找出这些格子是一个好想法。在 expand1 中实现这个想法的定义如下：

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
expand1 rows
 = [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
   where
   (rows1,row:rows2) = break (any smallest) rows
   (row1,cs:row2)    = break smallest row
   smallest cs       = length cs == n
   n                 = minimum (counts rows)
```

103

函数 counts 定义为

```
counts = filter (/= 1) . map length . concat
```

值 n 是选择矩阵中不等于 1 的最小选择数。函数 minimum 的定义将留作练习。如果矩阵中包含任何选择为空的元素，那么 n 将会是 0，此时 expand1 将返回一个空列表。另一方面，如果矩阵只含单个选择元素，那么 n 将是空列表的最小值，这是无定义值 ⊥，此时 expand1 也将返回 ⊥，所以最好确保 expand1 被应用于非完全矩阵。如果一个矩阵不满足 complete，则是非完全的：

```
complete :: Matrix [Digit] -> Bool
complete = all (all single)
```

也可以将 valid 推广到选择矩阵上的测试。假定如下定义 safe：

```
safe :: Matrix [Digit] -> Bool
safe m = all ok (rows m) &&
         all ok (cols m) &&
         all ok (boxs m)
ok row = nodups [x | [x] <- row]
```

如果一个矩阵的任何行、列和块中的单选择都不重复，则称矩阵是安全的（safe）。但是，一个矩阵可能包含非单选择列表元素。裁剪可以使得安全矩阵变为不安全矩阵，但是如果一个矩阵裁剪后是安全的，那么裁剪前一定是安全的，用公式表示为 safe . prune = safe。一个完全而且安全的矩阵便是数独谜题的一个解，而且这个解可以用简化的 expand 抽取：

```
extract :: Matrix [Digit] -> Grid
extract = map (map head)
```

因此，在安全与完全的矩阵 m 上有

```
filter valid (expand m) = [extract m]
```

在安全但不完全的矩阵上有

```
  filter valid . expand
= filter valid . concat . map expand . expand1
```

在两边进行置换的情况下等号成立。因为

104

```
filter p . concat = concat . map (filter p)
```

因此 filter valid . expand 化简为

```
concat . map (filter p . expand) . expand1
```

现在可以插入一个裁剪，得到

```
concat . map (filter p . expand . prune) . expand1
```

因此，引入下列函数：

```
search = filter valid . expand . prune
```

在安全但不完全的矩阵上有

```
search = concat . map search . expand1 . prune
```

现在可以用下面的第三个版本代替 solve：

```
solve = search . choices
search cm
  | not (safe pm) = []
  | complete pm   = [extract pm]
  | otherwise     = concat (map search (expand1 pm))
  where pm = prune cm
```

这是最终的数独简单求解器。可以在最后一行将 prune 用 many prune 代替，有时多次裁剪比一次裁剪快，有时则不然。注意到第一次安全检查是紧跟在扩展选择的一轮裁剪后，因此有问题的输入会很快被发现。

5.5 习题

习题 A 如何在整数矩阵上给每个元素加 1？如何求一个矩阵所有元素之和？函数 zipWith (+) 将两行相加，那么什么函数可以将两个矩阵相加？如何定义矩阵乘积？

习题 B 请问矩阵[[],[]]的维数是什么？矩阵[]呢？

105 函数 cols（这里重命名为 transpose）的定义是

```
transpose :: [[a]] -> [[a]]
transpose [xs]     = [[x] | x <- xs]
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

请填写下面的省略部分，然后可用它作为上面定义的第一个子句。

```
transpose []       = ...
```

以上定义的 transpose 按行进行。下面是按列进行转置的部分定义：

```
transpose xss = map head xss:transpose (map tail xss)
```

请完成该定义。

习题 C 下列哪些等式成立（不必证明）：

```
any p = not . all (not p)
any null = null . cp
```

习题 D 给定一个列表排序函数 sort :: (Ord a) => [a] -> [a]，请给出下列函数的定义。

```
nodups :: (Ord a) => [a] -> Bool
```

习题 E 函数 nub :: (Eq a) => [a] -> [a]删除列表中的重复元素（该函数的一个版本见模块 Data.List）。请给出 nub 的定义。假定结果中元素的次序不重要，请定义：

```
nub :: (Ord a) => [a] -> [a]
```

使得该函数更高效。

习题 F 函数 takeWhile 和 dropWhile 满足下列等式：

```
span p xs = (takeWhile p xs,dropWhile p xs)
```

请使用直接递归定义 takeWhile 和 dropWhile。

假定 whiteSpace :: Char -> Bool 测试一个字符是空白（如空格、tab 键和换行符）还是可见字符，请给出下列函数的定义。 106

```
words :: String -> [Word]
```

该函数将一个串拆分成词的列表。

习题 G 请定义 minimum :: Ord a => [a] -> a。

习题 H 为什么没有如下定义 solve?

```
solve = search . choices
search m
 | not (safe m) = []
 | complete m   = [extract m]
 | otherwise    = process m
 where process = concat . map search . expand1 . prune
```

5.6 答案

习题 A 答案 给矩阵每个元素加 1 可定义为 map (map (+1))。

对矩阵元素求和可以用 sum . map sum 实现，其中 sum 对数的列表求和。另一种方法是用 sum . concat。

矩阵相加定义为 zipWith (zipWith (+))。

对于矩阵乘积，可以先定义：

```
scalarMult :: Num a => [a] -> [a] -> a
scalarMult xs ys = sum (zipwith (*) xs ys)
```

然后定义：

```
matMult :: Num a => Matrix a -> Matrix a -> Matrix a
matMult ma mb = [map (scalarMult row) mbt | row <- ma]
                where mbt = transpose mb
```

107

习题 B 答案 矩阵[[],[]]的维数是 2×0。矩阵[]的维数是 $0\times n$，n 是任意整数。这种矩阵的转置必须具有维数 $n\times 0$，n 是任意整数。唯一的可能是令 n 为无穷大：

```
transpose :: [[a]] -> [[a]]
transpose []       = repeat []
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

其中 repeat x 返回 x 的无穷次重复列表。注意到

```
transpose [xs] = zipWith (:) xs (repeat [])
               = [[x] | x <- xs]
```

另一种定义是

```
transpose ([]:xss) = []
transpose xss = map head xss:transpose (map tail xss)
```

第一行的假设是，如果第一行是空的，那么所有行是空的，因此转置是空矩阵。

习题 C 答案 两个等式均成立。

习题 D 答案

```
nodups :: (Ord a) => [a] -> Bool
nodups xs = and (zipWith (/=) ys (tail ys))
            where ys = sort xs
```

习题 E 答案

```
nub :: (Eq a) => [a] -> [a]
nub []     = []
nub (x:xs) = x:nub (filter (/= x) xs)

nub :: (Ord a) => [a] -> [a]
nub = remdups . sort

remdups []     = []
remdups (x:xs) = x:remdups (dropWhile (==x) xs)
```

108 函数 dropWhile 在习题 F 中定义。

习题 F 答案

```
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
          = if p x then x:takeWhile p xs else []
dropWhile p [] = []
dropWhile p (x:xs)
          = if p x then dropWhile p xs else x:xs
```

函数 words 的定义是

```
words :: String -> [Word]
words xs | null ys   = []
         | otherwise = w:words zs
         where ys = dropWhile whiteSpace xs
               (w,zs) = break whiteSpace ys
```

习题 G 答案

```
minimum :: Ord a => [a] -> a
minimum [x]    = x
minimum (x:xs) = x `min` minimum xs
```

注意空列表的最小值无定义。

习题 H 答案 如果一次裁剪后矩阵是完全的，那么 solve 的这个定义将返回无定义值。

5.7 注记

《卫报》不再使用本章开端关于数独的说明。本章内容来自我的《Pearls of Functional Algorithm Design》（Cambridge，2010）。以下网页包含大约 20 个数独的 Haskell 实现：

haskell. org/haskellwiki/Sudoku

109 许多实现使用数组或者单子。我们将在第 10 章讨论数组和单子。

证　明

我们已经在前两章看到许多定律，尽管“定律”这个词有点不恰当，因为它意味着它们是从天上来的，无需证明。不过至少简短是这个词的优点。目前遇到的定律均断言两个函数表达式的相等（可能在某些附加条件下），换句话说，定律的形式是函数之间的等式或者恒等式（identity），并且计算是点自由式的计算（关于点自由式计算参见第4章及其习题K的答案）。给定适当的定律，可以利用等式推理证明其他定律。等式逻辑是函数式程序设计中简单且有效的工具，因为它能引导人们找到已构造的函数或者值的新的而且更有效的定义。效率是第7章的主题。本章是关于等式推理的另一方面，即归纳证明。本章还将引入能够描述计算的共同特征的高阶（higher order）函数，说明如何用高阶函数简化证明。可以证明这些高阶函数的一般性质，然后将其应用于其他函数，而不必对相似的函数性质一次次重复证明。

6.1　自然数上的归纳法

考虑下面的幂函数定义：

```
exp :: Num a => a -> Nat -> a
exp x Zero     = 1
exp x (Succ n) = x * exp x n
```

以前的定义可能是

110

```
exp :: Num a => a -> Int -> a
exp x 0     = 1
exp x (n+1) = x * exp x n
```

但是目前Haskell的标准版本Haskell 2010中不再允许使用这种（$n+1$）模式的定义。

无论如何定义，下列等式对任意m和n都应该成立。

```
exp x (m+n) = exp x m * exp x n
```

因为数学等式$x^{m+n} = x^m x^n$是成立的。但是如何证明这个定律呢？答案当然是*归纳法*（induction）。每个自然数或者是`Zero`或者形如`Succ n`，其中n是某个自然数。这也正是数据类型`Nat`的定义描述的：

```
data Nat = Zero | Succ Nat
```

所以，要证明$P(n)$对于任意自然数n成立，只要证明：

1. $P(0)$成立；
2. 对于任意自然数n，假定$P(n)$成立，则$P(n+1)$成立。

这里已经恢复使用0表示`Zero`，$n+1$表示`Succ n`，今后也将沿用这种记法。在第二个证明中，可以假定$P(n)$成立，并在证明$P(n+1)$时使用该假设。

作为例子，证明对于所有x、m和n下列等式成立。

```
exp x (m+n) = exp x m * exp x n
```

现在是对 m 归纳，也可以对 n 归纳，但是证明会更复杂。以下是证明。

0 的情况

```
 exp x (0 + n)
=  {因为 0 + n = n}
 exp x n
```

```
 exp x 0 * exp x n
=  {exp.1}
 1 * exp x n
=  {因为 1 * x = x}
 exp x n
```

111

$m+1$ 的情况

```
 exp x ((m + 1) + n)
=  {算术运算}
 exp x ((m + n) + 1))
=  {exp.2}
 x * exp x (m + n)
=  {归纳假设}
 x * (exp x m * exp x n)
```

```
 exp x (m+1) * exp x n
=  {exp.2}
 (x * exp x m) * exp x n
=  {因为 * 满足结合律}
 x * (exp x m * exp x n)
```

以上证明格式将会用于归纳证明。证明分两种情况：基本情况（base case）0 和归纳情况（induction case）$n+1$。每种情况分两列，一列对应等式左边，另一列对应等式右边。（若两列太宽写不下时，则一列接一列写。）化简等式每边表达式，直至两边化简到同一个表达式，即可完成证明的每种情况。括号中的理由 `exp.1` 和 `exp.2` 表示 `exp` 定义中的第一个和第二个等式。

最后，注意到证明应用了 3 个定律，即

```
(m + 1) + n = (m + n) + 1
1 * x       = x
(x * y) * z = x * (y * z)
```

如果从零开始重建算术——非常冗长的过程，那么这些定律也需要证明。事实上，只有第一个定律可以证明，因为它只涉及自然数，而且已定义了加法运算。后两个定律依赖于 Haskell 中类族 `Num` 的各种实例中乘法的实现。

实际上，结合律最终转换为浮点数的结合律：

```
ghci> (9.9e10 * 0.5e-10) * 0.1e-10 :: Float
4.95e-11
ghci> 9.9e10 * (0.5e-10 * 0.1e-10) :: Float
4.9499998e-11
```

回顾科学记法的 `9.9e10` 表示 9.9×10^{10}。所以，尽管证明在数学上正确，但是其中的一个等式至少在 Haskell 中是不正确的。

112

6.2 列表归纳法

每个有穷列表或者是空列表 `[]`，或者形如 `x:xs`，其中 `xs` 是一个有穷列表。所以，要证明 P(`xs`) 对于所有有穷列表 `xs` 成立，需要证明：

1. P(`[]`) 成立；
2. 对于任意 `x` 和任意有穷列表 `xs`，假设 P(`xs`) 成立，则 P(`x:xs`) 成立。

作为例子，回顾串联（++）的定义：

```
[] ++ ys     = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

现在证明++满足分配律，即对于所有有穷列表xs，所有列表ys和zs（注意不要求后两个列表是有穷的），下列等式成立。

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

证明对xs使用归纳法：

[]的情况

```
 ([] ++ ys) ++ zs            [] ++ (ys ++ zs)
=  {++.1}                  =  {++.1}
 ys ++ zs                   ys ++ zs
```

x:xs的情况

```
 ((x:xs) ++ ys) ++ zs          (x:xs) ++ (ys ++ zs)
=  {++.2}                    =  {++.2}
 (x:(xs ++ ys)) ++ zs         x:(xs ++ (ys ++ zs))
=  {++.2}                    =  {归纳假设}
 x:((xs ++ ys) ++ zs)         x:((xs ++ ys) ++ zs)
```

再看一个例子。给定下列定义：

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

证明reverse是一个对合，即对于所有有穷列表xs下列等式成立。

```
reverse (reverse xs) = xs
```

递归的基本情况简单，容易验证。归纳情况证明如下：

113

x:xs的情况

```
 reverse (reverse (x:xs))
=  {reverse.2}
 reverse (reverse xs ++ [x])
=  {????}
 x:reverse (reverse xs)
=  {归纳假设}
 x:xs
```

等式右边的化简省略了，因为它本身就是x:xs。但是上面的证明过程中间被卡住了。这里需要一个辅助结果，即对于任意有穷列表ys下列等式成立。

```
reverse (ys ++ [x]) = x:reverse ys
```

这个辅助命题也可以用归纳法证明：

[]的情况

```
 reverse ([] ++ [x])          x:reverse []
=  {++.1}                   =  {reverse.1}
 reverse [x]                  [x]
=  {reverse.2}
```

```
  reverse [] ++ [x]
=   {reverse.1 and ++.1}
  [x]
```

y:ys 的情况

```
  reverse ((y:ys) ++ [x])
=   {++.2}
  reverse (y:(ys ++ [x]))
=   {reverse.2}
  reverse (ys ++ [x]) ++ [y]
=   {归纳假设}
  (x:reverse ys) ++ [y]
=   {++.2}
  x:(reverse ys ++ [y])
```

```
  x:reverse (y:ys)
=   {reverse.2}
  x:(reverse ys ++ [y])
```

114

这就证明了辅助命题成立，所以主命题成立。

非完整列表上的归纳

非完整列表或者是无定义列表，或者形如 x:xs，其中 xs 是非完整列表。因此，要证明 P(xs) 对于所有非完整列表成立，需要证明：

1. P(undefined) 成立；
2. 对于任意 x 和任意非完整列表 xs，假定 P(xs) 成立，则 P(x:xs) 成立。

作为例子，下面证明对于任意非完整列表 xs 和任意列表 ys 下列等式成立。

```
xs ++ ys = xs
```

undefined 的情况

```
  undefined ++ ys
=   {++.0}
  undefined
```

x:xs 的情况

```
  (x:xs) ++ ys
=   {++.2}
  x:(xs ++ ys)
=   {归纳假设}
  x:xs
```

每种情况中省略了等式右边的平凡化简。理由（++.0）表示（++）定义中失败的子句：因为串联是对左边参数模式匹配定义的，如果该参数无定义，则结果也无定义。

无穷列表上的归纳

115

证明无穷列表的某个性质需要今后几章的内容做基础。无穷列表基本上可以理解为一个非完整列表序列的极限。例如，[0..]是下面序列的极限。

```
undefined,  0:undefined,  0:1:undefined,  0:1:2:undefined,
```

如果一个性质 P 满足，对于任意以 xs 为极限的序列 xs_0，xs_1，…，当 $P(xs_n)$ 成立

（对于所有 n）时 $P(xs)$ 也成立，则称性质 P 为**链完全的**（chain complete）。

换句话说，如果 P 是链完全的性质，而且对于所有非完整列表（也可能包含所有有穷列表）成立，则性质 P 对所有无穷列表成立。

许多性质是链完全的，例如：

- 所有等式 e1 = e2 是链完全的，其中 e1 和 e2 是包含受全称量词约束自由变量的 Haskell 表达式；
- 如果 P 和 Q 都是链完全的，那么它们的合取 $P \wedge Q$ 是链完全的。

但是，不等式 e1 ≠ e2 不一定是链完全的，同样存在量词表示的性质也不一定是链完全的。例如，考虑断言：存在某个整数使得

```
drop n xs = undefined
```

这个性质对于所有的非完整列表显然是成立的，对于无穷列表显然不成立。

下面是一个证明的例子。前面曾证明了对于所有的有穷列表 xs 和所有列表 ys 和 zs，下列等式成立。

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

可以证明，这个链完全的性质可以推广到所有的列表 xs。

undefined 的情况

```
  (undefined ++ ys) ++ zs          undefined ++ (ys ++ zs)
=   {++.0}                     =   {++.0}
  undefined ++ zs                  undefined
=   {++.0}
  undefined
```

所以，++ 是列表上真正的可结合运算，不论列表是有穷的、非完整的还是无穷的。

不过推广性质要格外小心。前面曾经证明了对于所有有穷列表有 116

```
reverse (reverse xs) = xs
```

通过证明下面的额外情况，能否将等式推广到所有列表？

undefined 的情况

```
  reverse (reverse undefined)
=   {reverse.0}
  undefined
```

这种情况证明通过，但是等式仍然有问题，对于任意非完整列表 xs，得到 Haskell 等式：

```
reverse (reverse xs) = undefined
```

是哪里出了问题？答案是，证明 reverse 的对合性用到了对于任意有穷列表 ys 的下列辅助结果：

```
reverse (ys ++ [x]) = x:reverse ys
```

这个等式并不是对所有列表成立，而且对于非完整列表 ys 确实不成立。

由此得出 reverse 不是列表上的恒等函数。列表上的函数等式 f = g 表示对于所有

列表 xs，即有穷、非完整和无穷列表，等式 f xs = g xs 都成立。如果等式仅对有穷列表成立，则必须明确说明。

6.3 函数 foldr

下面的函数都有一个共同的模式：

```
sum []     = 0
sum (x:xs) = x + sum xs

concat []       = []
concat (xs:xss) =  xs ++ concat xss

filter p []     = []
filter p (x:xs) = if p x then x:filter p xs
                  else filter p xs

map f []     = []
map f (x:xs) = f x:map f xs
```

117

类似地，下列定律的归纳证明都有一个共同的模式：

```
sum (xs ++ ys)      = sum xs + sum ys
concat (xss ++ yss) = concat xss ++ concat yss
filter p (xs ++ ys) = filter p xs ++ filter p ys
map f (xs ++ ys)    = map f xs ++ map f ys
```

难道不能将以上函数定义为一个更通用函数的特例，同样将以上定律定义为一个通用定律的特例？这样将会节省大量的重复劳动。

函数 foldr（从右边折叠）定义如下：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

为理解定义，考虑下例：

```
foldr (@) e [x,y,z] = x @ (y @ (z @ e))
            [x,y,z] = x : (y : (z : []))
```

换句话说，foldr (@) e 应用于一个列表的结果是将空列表用 e 代替，将(:)用(@)代替，然后对表达式求值。括号是右结合的，如函数名所指。

由此即得，foldr (:) []是列表上的恒等函数。更多的结果如下：

```
sum      = foldr (+) 0
concat   = foldr (++) []
filter p = foldr (\x xs -> if p x then x:xs else xs) []
map f    = foldr ((:) . f) []
```

下列事实抓住了前面提到的各种等式：

```
foldr f e (xs ++ ys) = foldr f e xs @ foldr f e ys
```

其中(@)是某个满足不同性质的运算。下面对 xs 进行归纳证明该等式。在证明过程中可以发现 f、e 和(@)需要满足的性质。

118

[]的情况

```
 foldr f e ([] ++ ys)       foldr f e [] @ foldr f e ys
=  {++.1}                 =  {foldr.1}
 foldr f e ys               e @ foldr f e ys
```

所以，需要的条件是，对于任意 x，等式 e @ x = x 成立。

x:xs 的情况

```
 foldr f e ((x:xs) ++ ys)
=  {++.2}
 foldr f e (x:(xs ++ ys)
=  {foldr.2}
 f x (foldr f e (xs ++ ys))
=  {归纳假设}
 f x (foldr f e xs @ foldr f e ys)
```

这种情况的右边简化为

```
f x (foldr f e xs) @ foldr f e ys
```

所以，总的来说，需要下列等式：

```
e @ x       = x
f x (y @ z) = f x y @ z
```

对于任意 x、y 和 z 成立。特别是，如果 f = (@)并且(@)是与单位元 e 结合的，那么以上要求即可满足。由此即可证明：

```
sum (xs ++ ys)      = sum xs + sum ys
concat (xss ++ yss) = concat xss ++ concat yss
```

对于 map 的定律，需要下面等式：

```
[] ++ xs = xs
f x:(xs ++ ys) = (f x:xs) ++ ys
```

这两个等式可以根据串联定义立即证明。

对于 filter 的定律需要满足

```
if p x then x:(ys ++ zs) else ys ++ zs
  = (if p x then x:ys else ys) ++ zs
```

119

根据串联和条件表达式定义，以上等式也成立。

融合

函数 foldr 最重要的性质是融合定律（fusion law）：

```
f . foldr g a =  foldr h b
```

只要其中的元素满足一定的性质。下面是两个简单的例子：

```
double . sum    = foldr ((+) . double) 0
length . concat = foldr ((+) . length) 0
```

实际上，已看到的许多定律是 foldr 融合律的特例。总之，融合律是列表上归纳法的预制包。

至于融合律中需要什么条件，可以在融合律的归纳证明过程中发现。融合律是一个函数等式，所以需要证明等式对于所有有穷列表和非完整列表成立。

undefine 的情况

```
 f (foldr g a undefined)          foldr h b undefined
=  {foldr.0}                    =  {foldr.0}
 f undefined                      undefined
```

所以，第一个条件是，f 必须是严格的。

[]的情况

```
 f (foldr g a [])              foldr h b []
=  {foldr.1}                 =  {foldr.1}
 f a                           b
```

第二个条件是 f a = b。

x:xs 的情况

```
 f (foldr g a (x:xs))          foldr h b (x:xs)
=  {foldr.2}                  =  {foldr.2}
 f (g x (foldr g a xs))        h x (foldr h b xs)
                              =  {归纳假设}
                               h x (f (foldr g a xs))
```

120

第三个条件是 f (g x y) = h x (f y)对任意 x 和 y 成立。

现在利用融合律来证明下列等式：

```
foldr f a . map g = foldr h a
```

注意到 map f = foldr ((:) . g)。检查融合律的条件，有下列等式：

```
foldr f a undefined = undefined
foldr f a []        = a
```

所以，前两个融合条件满足。第三个条件是

```
foldr f a (g x:xs) = h x (foldr f a xs)
```

左边化简为

```
f (g x) (foldr f a xs)
```

所以，定义 h x y = f (g x) y，更简洁的表示是 h = f . g。因此，得到有用的规则：

```
foldr f a . map g = foldr (f . g) a
```

特别是

```
double . sum = sum . map double
             = foldr ((+) . double) 0

length . concat = sum . map length
                = foldr ((+) . length) 0
```

融合律的其他简单结论将在习题中讨论。

一种变形

有时候处理列表是很头疼的事。例如，空列表的最小元素是什么？为此，Haskell 提

供了 foldr 的另一种形式，称为 foldr1，该函数仅定义在非空列表上。函数 foldr1 的 Haskell 定义如下：

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

121

因此，可以定义：

```
minimum, maximum :: Ord a => [a] -> a
minimum = foldr1 min
maximum = foldr1 max
```

而且避免了两个显式递归。事实上，这里 foldr1 的定义不是它的最通用定义，这个问题留在习题中讨论。

6.4　函数 foldl

回顾等式

```
foldr (@) e [w,x,y,z] = w @ (x @ (y @ (z @ e)))
```

有时候对右边更方便的模式是

```
(((e @ w) @ x) @ y) @ z
```

这种计算模式由函数 foldl（从左边折叠）概括如下：

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

例如，假定已知一个串，如表示一个实数 1234.567 的串，需要求出它的整数部分和小数部分。可以定义：

```
ipart :: String -> Integer
ipart xs = read (takeWhile (/= '.') xs) :: Integer

fpart :: String -> Float
fpart xs = read ('0':dropWhile (/= '.' xs) :: Float
```

其中使用了类族 Read 的方法 read。顺便指出，.567 不是 Haskell 的合式文字。为了避免与函数复合运算符混淆，在小数点前后都至少应该有一个数字。例如：

```
ghci> :t 3 . 4
3 . 4 :: (Num (b -> c), Num (a -> b)) => a -> c
```

另外一种定义是

122

```
parts :: String -> (Integer,Float)
parts ds = (ipart es,fpart fs)
           where (es,d:fs) = break (== '.') ds
ipart    = foldl shiftl 0 . map toDigit
           where shiftl n d = n*10 + d
fpart    = foldr shiftr 0 . map toDigit
           where shiftr d x = (d + x)/10
toDigit d = fromIntegral (fromEnum d - fromEnum '0')
```

因为

$$
\begin{aligned}
1234 &= 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \\
&= (((0 \times 10 + 1) \times 10 + 2) \times 10 + 3) \times 10 + 4 \\
0.567 &= 5/10 + 6/100 + 7/1000 \\
&= (5 + (6 + (7 + 0)/10)/10)/10
\end{aligned}
$$

所以，使用 foldl 取得整数部分和使用 foldr 得到小数部分都得到了展示。

再如，函数 reverse 由如下等式定义：

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

我们现在学到了更聪明的方法，可以定义：

```
reverse = foldr snoc []
          where snoc x xs = xs ++ [x]
```

但是，一知半解是危险的：reverse 的两个定义都很糟，因为求长度为 n 的列表的逆均需要 n^2 步。更好的定义是

```
reverse = foldl (flip (:)) []
```

其中 flip f x y = f y x。现在列表求逆的新定义是线性的：

```
  foldl (flip (:)) [] [1,2,3]
= foldl (flip (:)) (1:[]) [2,3]
= foldl (flip (:)) (2:1:[]) [3]
= foldl (flip (:)) (3:2:1:[]) []
= 3:2:1:[]
```

123 计算过程似乎有点复杂，但是这个新定义涉及一个完整的工作原理，将在第 7 章介绍。

如本例所示，函数 foldr 和 foldl 具有下列关系，对于任意有穷列表 xs 有

```
foldl f e xs = foldr (flip f) e (reverse xs)
foldr f e xs = foldl (flip f) e (reverse xs)
```

证明留作习题。注意对有穷列表的限制，当 xs 为 $\perp$ 时，等式两边均为 $\perp$。这也表明，证明必须依赖于一个只对有穷列表成立的辅助性质。

下面是函数 foldr 和 foldl 的另一种关系：对任意有穷列表 xs，如果

```
(x <> y) @ z = x <> (y @ z)
e @ x        = x <> e
```

则有

```
foldl (@) e xs = foldr (<>) e xs
```

证明留作习题。作为这个定律的应用演示，假定 (<>) = (@)，而且 (@) 与单位元 e 可结合。那么两个条件都成立，所以可以断言：对于任意有穷列表，如果 (@) 与单位元 e 可结合，那么

```
foldr (@) e xs = foldl (@) e xs
```

特别是，对于任意有穷列表 xss，下列等式成立：

```
concat xss = foldr (++) [] xss = foldl (++) [] xss
```

如果 xss 是无穷列表，那么两个定义是不同的：

```
ghci> foldl (++) [] [[i] | i <- [1..]]
Interrupted.
ghci> foldr (++) [] [[i] | i <- [1..]]
[1,2,3,4,{Interrupted}
```

对于第一个表达式，GHCi 陷入长久沉默，计算被“终止程序运行”键中断。对于第二个表达式，GHCi 开始输出一个无穷列表。所以，使用 foldr 的定义在无穷列表上可行，但是另一个不可行。但是，或许使用 foldl 定义的 concat 在所有列表有穷时效率更高。为了回答这个问题，观察下面的计算：

```
foldr (++) [] [xs,ys,us,vs]
          = xs ++ (ys ++ (us ++ (vs ++ [])))
foldl (++) [] [xs,ys,us,vs]
          = ((((] ++ xs) ++ ys) ++ us) ++ vs)
```

124

假设以上计算中每个列表的长度都是 n，那么右边的第一个表达式需要 $4n$ 步完成所有串联，而第二个表达式需要 $0+n+(n+n)+(n+n+n)=6n$ 步才能完成所有串联。这足以说明问题，至少到目前为止。

6.5 函数 scanl

函数 scanl f e 将 foldl f e 应用于一个列表的每个前缀。例如：

```
ghci> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
```

该表达式计算列表前 10 个正整数的*流动和*（running sum）：

```
[0, 0+1, (0+1)+2, ((0+1)+2)+3, (((0+1)+2)+3)+4, ...]
```

函数 scanl 的说明是

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl f e = map (foldl f e) . inits

inits :: [a] -> [[a]]
inits []     = [[]]
inits (x:xs) = [] : map (x:) (inits xs)
```

例如：

```
ghci> inits "barbara"
["","b","ba","bar","barb","barba","barbar","barbara"]
```

函数 inits 在模块 Data.List 中定义。

但是，在长度为 n 的列表上，scanl f 的这个定义计算 f 的次数是

$$0+1+2+\cdots+n=n(n+1)/n$$

能给出更好的定义吗？是的，通过某种归纳证明可以找到一种更好的定义，只是不清楚要证明的是什么！

125

[]的情况

```
 scanl f e []
=  {定义}
 map (foldl f e) (inits [])
=  {inits.1}
 map (foldl f e) [[]]
```

```
=  {map.1 和 map.2}
 [foldl f e []]
=  {foldl.1}
 [e]
```

因此，已经证明了 `scanl f e [] = [e]`。

x:xs 的情况

```
 scanl f e (x:xs)
=  {定义}
 map (foldl f e) (inits (x:xs))
=  {inits.2}
 map (foldl f e) ([]:map (x:) (inits xs))
=  {map.1 和 map.2}
 foldl f e []:map (foldl f e . (x:)) (inits xs)
=  {foldl.1}
 e:map (foldl f e . (x:)) (inits xs)
=  {要求:foldl f e . (x:) = foldl f (f e x)}
    e:map (foldl f (f e x)) (inits xs)
=  {scanl 的定义}
 e:scanl f (f e x) xs
```

证明中要求的条件是 `foldl` 的显然结论。因此，总结起来，已经证明了：

```
scanl f e []     = [e]
scanl f e (x:xs) = e:scanl f (f e x) xs
```

126 这个定义只需要计算线性次数的 `f`。

以上所做的是通过程序计算（program calculation）优化函数的一个例子。Haskell 令人兴奋的一点就是完全可以完成这样的任务，而不必另外引入一种不同的逻辑语言对程序推理。

不过，引导库中 `scanl` 的定义稍有不同：

```
scanl f e xs = e : (case xs of
                      []   -> []
                      x:xs -> scanl f (f e x) xs)
```

在我们的定义中 `scanl f e undefined = undefined`，而对于引导库函数定义有 `scanl f e undefined = e:undefined`.

原因是在定义中，两个子句的右边都是 `e` 开始的列表，对这个事实无需知道左边是什么，而且惰性计算令人无需多问。

库函数定义还使用了 `case` 表达式。本书极少用这种表达式，故不做详细介绍。Haskell 允许用许多方法表达同一件事。

6.6 最大连续段和问题

下面是另一个程序计算的例子。最大段和（maximum segment sum）问题是一个有名的问题，其历史在 J. Bentley 的《Programming Pearls》（1987）中有描述。给定一个整数序列，要求计算序列中所有段（segment）的和的最大值。一个段也称为一个连续子序列

(contiguous subsequence)。例如，对序列 [-1, 2, -3, 5, -2, 1, 3, -2, -2, -3, 6]，其最大段和是7，它是 [5, -2, 1, 3] 的和。另一方面，序列 [-1, -2, -3] 的最大段和是0，因为空序列是每个列表的段，其和是0。由此可得，最大段和总是非负的。

这个问题可以如下说明：

```
mss  :: [Int] -> Int
mss = maximum . map sum . segments
```

127

其中 segments 返回一个列表的所有段构成的列表。这个函数有多种定义方法，包括：

```
segments = concat . map inits . tails
```

其中 tails 是 inits 的对偶，它返回一个列表的所有尾段：

```
tails :: [a] -> [[a]]
tails []     = [[]]
tails (x:xs) = (x:xs):tails xs
```

segments 的定义描述了取得所有尾段的首段的过程。例如：

```
ghci> segments "abc"
["","a","ab","abc","","b","bc","","c",""]
```

其中空列表出现了4次，对每个尾段出现一次。

在长度为 n 的列表上直接计算 mss 需要的步数正比于 n^3。因为总共有 n^2 个段，每个段求和需要 n 步，所有总共需要 n^3 步。求解这个问题是否可以做得比三次方更好，没有明显的答案。

但是，看看程序计算会给出什么样的指引。首先安装 segments 的定义：

```
maximum . map sum . concat . map inits . tails
```

查看可以使用的定律，发现

```
map f . concat = concat . map (map f)
```

可以应用到中间的项 map sum . concat。由此得到

```
maximum . concat . map (map sum) . map inits . tails
```

现在可以使用定律 map f . map g = map (f . g)，得到

```
maximum . concat . map (map sum . inits) . tails
```

对了，还可以使用定律：

```
maximum . concat = maximum . map maximum
```

是不是？不是的，除非 concat 的参数是非空列表的非空列表，因为空列表的最大值是没有定义的。对于目前的例子，这个定律成立，因为 inits 和 tails 均返回非空列表。由此得到

128

```
maximum . map (maximum . map sum . inits) . tails
```

下一步将使用6.5节描述的 scanl 的性质，即

```
map sum . inits =  scanl (+) 0
```

由此得到

```
maximum . map (maximum . scanl (+) 0) . tails
```

现在我们已经将 n^3 算法简化到 n^2 算法，所以已经取得了进展。现在看似卡住了，因为在我们的兵器库里没有定律可用。

下一步明显地有关 maximum . scanl (+) 0。所以，先来看看对下列式子能证明什么。

```
foldr1 max . scanl (+) 0
```

这个式子看起来像融合律，但是 scanl (+) 0 能够用 foldr 表达吗？是的，例如：

```
  scanl (+) 0 [x,y,z]
= [0,0+x,(0+x)+y,((0+x)+y)+z]
= [0,x,x+y,x+y+z]
= 0:map (x+) [0,y,y+z]
= 0:map (x+) (scanl (+) 0 [y,z])
```

这些计算显示了(+)的结合律以及 0 是(+)的单位元。更一般地，这些结果提示我们只要(@)是可结合的，有单位元 e，那么

```
scanl (@) e = foldr f [e]
  where f x xs = e:map (x@) xs
```

让我们承认这个定律，继续寻找使得下式成立的条件：

```
foldr1 (<>) . foldr f [e] = foldr h b
  where f x xs = e:map (x@) xs
```

显然 foldr1 (<>)是严格的，而且 foldr1 (<>) [e] = e，故有 b = e。接下来需要检查融合律的第三个条件是否满足：对所有 x 和所有 xs，需要 h 满足

```
foldr1 (<>) (e:map (x@) xs) = h x (foldr1 (<>) xs)
```

129 等式的左边化简为

```
e <> (foldr1 (<>) (map (x@) xs))
```

取单元素的情况 xs = [y]，发现

```
h x y = e <> (x @ y)
```

这个结果给出 h 的定义，但是仍然需要检查下列等式是否成立。

```
foldr1 (<>) (e:map (x@) xs) = e <> (x @ foldr1 (<>) xs)
```

将等式两边化简，该等式成立的条件是

```
foldr1 (<>) . map (x@) = (x@) . foldr1 (<>)
```

最后一个等式成立的条件是(@)对于(<>)可分配，也就是

```
x @ (y <> z) = (x @ y) <> (x @ z)
```

证明留作习题。

加法对（二元）取最大运算可分配吗？是的：

```
x + (y `max` z) = (x + y) `max` (x + z)
x + (y `min` z) = (x + y) `min` (x + z)
```

再返回到最大段和问题。已经得到了

```
maximum . map (foldr (@) 0) . tails
where x @ y = 0 `max` (x + y)
```

得到的这个结果看似很像6.5节scanl定律的一个特例，只是这里使用了foldr而不是foldl，还使用了tails，而不是inits。但是，进行与scanl类似的计算显示

```
map (foldr f e) . tails = scanr f e
```

其中：

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f e []     = [e]
scanr f e (x:xs) = f x (head ys):ys
                   where ys = scanr f e xs
```

函数scanr在标准引导库也有定义。总之，有

```
mss = maximum . scanr (@) 0
      where  x @ y = 0 `max` (x + y)
```

该结果是求最大段和的线性时间解。 130

6.7 习题

习题A 第3章定义了自然数乘法。下面是稍有区别的定义：

```
mult :: Nat -> Nat -> Nat
mult Zero y     = Zero
mult (Succ x) y = mult x y + y
```

请证明mult (x+y) z = mult x z + mult y z。证明只能利用x+0=x和(+)满足结合律的事实。所以，需要好好考虑，3个变量x、y和z中，在哪个变量上归纳最好。

习题B 证明：对于所有的有穷列表xs和ys，有

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

可以假设(++)满足结合律。

习题C 还记得在第2章习题D中出现的朋友Eager Beaver和Lazy Susan吗？Susan喜欢使用head . map f，而Beaver更中意f . head。等一等！这两个表达式相等吗？请用归纳证明验证。

习题D 回顾第5章的笛卡儿积函数cp :: [[a]] -> [[a]]。用适当的f和e给出形如cp = foldr f e的定义。如果读者愿意的话，可以使用列表概括表示函数f。

本习题接下来的任务有关下面恒等式的证明：

```
length . cp = product . map length
```

其中product返回一个数值列表元素的乘积。

1. 使用融合定律将length . cp表示成foldr的特例。
2. 将map length表示成foldr的特例。 131
3. 再一次利用融合定律将product . map length表达成foldr的特例。
4. 检查两个结果应是相等的。如果不相等，那么cp定义有错误。

习题E foldr的前两个参数是列表的两个构造函数的替代物：

```
(:) :: a -> [a] -> [a]
[]  :: [a]
```

折叠函数可以在任何数据类型上定义：只需将数据类型的构造函数用参数代替即可。例如，考虑下面的数据类型：

```
data Either a b = Left a | Right b
```

要定义折叠函数，需要替换下列构造函数：

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

由此得到定义：

```
foldE :: (a -> c) -> (b -> c) -> Either a b -> c
foldE f g (Left x)  = f x
foldE f g (Right x) = g x
```

类型 Either 不是递归类型，所以 foldE 也不是递归函数。事实上，foldE 是一个标准引导库函数，只是函数名为 either，不是 foldE。

接着为下列类型定义折叠函数。

```
data Nat = Zero | Succ Nat
data NEList a = One a | Cons a (NEList a)
```

第二个定义引入非空列表。

请问 Haskell 的 foldr1 有什么问题?

习题 F 证明：对于任意有穷列表 xs，下列等式成立。

```
foldl f e xs = foldr (flip f) e (reverse xs)
```

132 再证明，如果下列条件成立：

```
(x <> y) @ z = x <> (y @ z)
e @ x        = x <> e
```

那么对于所有有穷列表 xs 下面等式成立：

```
foldl (@) e xs = foldr (<>) e xs
```

习题 G 利用等式：

```
foldl f e (xs ++ ys) = foldl f (foldl f e xs) ys
foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs
```

证明下列等式：

```
foldl f e . concat = foldl (foldl f) e
foldr f e . concat = foldr (flip (foldr f)) e
```

习题 H 在数学上，下列式子的值是什么?

```
sum (scanl (/) 1 [1..])
```

习题 I 由下面的说明计算 scanr 的有效定义。

```
scanr f e = map (foldr f e) . tails
```

习题 J 考虑下面的计算问题：

```
mss :: [Int] -> Int
mss = maximum . map sum . subseqs
```

其中 subseqs 返回一个列表的所有子序列，包括它本身：

```
subseqs :: [a] -> [[a]]
subseqs []     = [[]]
subseqs (x:xs) = xss ++ map (x:) xss
                 where xss =  subseqs xs
```

求 mss 的更高效定义。 133

习题 K　本习题的问题比较零散。

1. 函数 takePrefix p 应用于列表 xs，返回 xs 的满足 p 的最长前缀。因此

```
takePrefix :: ([a] -> Bool) -> [a] -> [a]
```

请问下列表达式的值是什么？

```
takePrefix nondec [1,3,7,6,8,9]
takePrefix (all even) [2,4,7,8]
```

请完成下列等式右边部分。

```
takePrefix (all p) = ...
```

请用标准函数，包括 inits 给出 takePrefix 的定义。

本题最后将会讨论 takePrefix。

2. 函数 one 和 ones 由下列方程定义：

```
one x  = [x]
none x = []
```

请完成下列恒等式的右边部分。

```
none . f     = ...
map f . none = ...
map f . one  = ...
```

3. 回顾定义 fork (f,g) x = (f x,g x)，完成下面恒等式。

```
fst . fork (f,g) = ...
snd . fork (f,g) = ...
fork (f,g) . h   = ...
```

4. 定义：

```
test p (f,g) x = if p x then f x else g x
```

请完成下列式子右边部分。

```
test p (f,g) . h  = ...
h . test p (f,g)  = ...
```

函数 filter 可以定义为 134

```
filter p = concat . map (test p (one,none))
```

使用以上恒等式以及其他标准恒等式，利用等式推理证明下列等式。

```
filter p = map fst . filter snd . map (fork (id,p))
```

（提示：像常见的计算一样，从较复杂的一边开始。）

5. 回顾第4章习题K答案中标准引导库函数curry和uncurry的定义：

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

请完成下式右边部分。

```
map (fork (f,g)) = uncurry zip . (??)
```

6. 再返回takePrefix，请利用等式推理计算下列表达式的有效程序。

```
takePrefix (p . foldl f e)
```

要求f的应用次数是线性的。

6.8 答案

习题A答案 证明对y进行归纳：

0的情况

```
 mult (x+0) z
=  {因为x + 0=x}
 mult x z
```

```
 mult x z + mult 0 z
=  {mult.1}
 mult x z + 0
=  {因为x + 0 = x}
 mult x z
```

135

y+1的情况

```
 mult (x+(y+1)) z
=  {因为(+)满足结合律}
 mult ((x+y)+1) z
=  {mult.2}
 mult (x+y) z + z
=  {归纳假设}
 (mult x z + mult y z) + z
```

```
 mult x z + mult (y+1) z
=  {mult.2}
 mult x z + (mult y z + z)
=  {因为(+)满足结合律}
 (mult x z + mult y z) + z
```

习题B答案 证明对xs归纳：

[]的情况

```
 reverse ([]++ys)
=  {++.1}
 reverse ys
```

```
 reverse ys ++ reverse []
=  {reverse.1}
 reverse ys ++ []
=  {因为xs ++ [] = xs}
 reverse ys
```

x:xs的情况

```
 reverse ((x:xs)++ys)
=  {++.2}
 reverse (x:(xs++ys))
=  {reverse.2}
 reverse (xs++ys) ++ [x]
=  {归纳假设}
 (reverse ys ++ reverse xs) ++ [x]
```

另一边的化简：

```
 reverse ys ++ reverse (x:xs)
=  {reverse.2}
 reverse ys ++ (reverse xs ++ [x])
=  {根据(++)的结合律}
 (reverse ys ++ reverse xs) ++ [x]
```

136

习题 C 答案　我们必须证明对于所有的列表（有穷、非完整和无穷列表）都有下列等式成立。

```
head (map f xs) = f (head xs)
```

对于 undefined 的情况和归纳情况 x:xs 容易验证，但是对于[]的情况，得到

```
head (map f []) = head [] = undefined
f (head [])     = f undefined
```

因此，该定律只有当 f 是严格函数时成立。Eager Beaver 不介意这个问题，因为它只构造严格函数。

习题 D 答案　可定义：

```
cp = foldr op [[]]
 where op xs xss = [x:ys | x <- xs, ys <- xss]
```

1. length . cp = foldr h b 成立的条件是 length 是严格的（是的），而且下列等式成立。

```
length [[]] = b
length (op xs xss) = h xs (length xss)
```

由第一个等式得出 b =1，因为

```
length (op xs xss) = length xs * length xss
```

由第二个等式得出 h = (*) . length。

2. map length = foldr f []，其中 f xs ns = length xs:ns。更短的定义是 f = (:) . length。

3. product . map length = foldr h b，只要 product 是严格的（是的）而且下列等式成立。

```
product [] = b
product (length xs:ns) = h xs (product ns)
```

由第一个等式得出 b = 1，因为

```
product (length xs:ns) = length xs * product ns
```

由第二个等式得出 h = (*) . length。

4. h 和 b 的定义是一样的。

137

习题 E 答案　foldN 的定义是直接的：

```
foldN :: (a -> a) -> a -> Nat -> a
foldN f e Zero     = e
foldN f e (Succ n) = f (foldN f e n)
```

特别是

```
m+n = foldN Succ m n
m*n = foldN (+m) Zero n
m^n = foldN (*m) (Succ Zero) n
```

对于非空列表，foldNE 的定义是

```
foldNE :: (a -> b -> b) -> (a -> b) -> NEList a -> b
foldNE f g (One x)      = g x
foldNE f g (Cons x xs) = f x (foldNE f g xs)
```

作为非空列表上的合理折叠，foldr1 的正确定义应该是

```
foldr1 :: (a -> b -> b) -> (a -> b) -> [a] -> b
foldr1 f g [x]    = g x
foldr1 f g (x:xs) = f x (foldr1 f g xs)
```

Haskell 的 foldr1 定义限制 g 为恒等函数。

习题 F 答案 为简洁起见，记 g = flip f。采用归纳法证明，对于任意的有穷列表 xs 有

```
foldl f e xs = foldr g e (reverse xs)
```

[]的情况

```
 foldl f e []
=  {foldl.1}
 e
```

```
 foldr g e (reverse [])
=  {reverse.1}
 foldr g e []
=  {foldr.1}
 e
```

138

x:xs 的情况

```
 foldl f e (x:xs)
=  {foldl.2}
 foldl f (f e x) xs
=  {归纳假设}
 foldr g (f e x) (reverse xs)
```

另一边的化简：

```
 foldr g e (reverse (x:xs))
=  {reverse.2}
 foldr g e (reverse xs ++ [x])
=  {需要的条件见下}
 foldr g (foldr g e [x]) (reverse xs)
=  {因为 foldr (flip f) e [x] = f e x}
 foldr g (f e x) (reverse xs)
```

需要的条件是

```
foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs
```

证明留给读者做练习。另外，一个相伴的结果是

```
foldl f e (xs ++ ys) = foldl f (foldl f e xs) ys
```

证明也留作练习。

采用归纳法证明，对于任意有穷列表 xs，下列等式成立。

```
foldl (@) e xs = foldr (<>) e xs
```

基本情况的证明容易。归纳情况的证明如下：

x:xs 的情况

```
 foldl (@) e (x:xs)                foldr (<>) e (x:xs)
=  {foldl.2}                     =  {foldr.2}
 foldl (@) (e @ x) xs              x <> foldr (<>) e xs
=  {假定 e @ x = x <> e}         =  {归纳假设}
 foldl (@) (x <> e) xs             x <> foldl (@) e xs
```

139

两边化简结果不同。需要另一个归纳假设：

```
foldl (@) (x <> y) xs = x <> foldl (@) y xs
```

基本情况的证明简单。归纳情况的证明如下：

z:zs 的情况

```
 foldl (@) (x <> y) (z:zs)
=  {foldl.2}
 foldl (@) ((x <> y) @ z) zs
=  {因为 (x <> y) @ z = x <> (y @ z)}
 foldl (@) (x <> (y @ z)) zs
=  {归纳假设}
 x <> foldl (@) (y @ z) zs
```

另一边的化简：

```
 x <> foldl (@) y (z:zs)
=  {foldl.2}
 x <> foldl (@) (y @ z) zs
```

习题 G 答案　用归纳法证明。基本情况的证明简单。归纳情况的证明如下：

```
 foldl f e (concat (xs:xss))
=  {concat 的定义}
 foldl f e (xs ++ concat xss)
=  {foldl 的给定性质}
 foldl f (foldl f e xs) (concat xss)
=  {归纳假设}
 foldl (foldl f) (foldl f e xs) xss
=  {foldl 的定义}
 foldl (foldl f) e (xs:xss)
```

140

另一边的化简：

```
 foldr f e (concat (xs:xss))
=  {concat 的定义}
 foldr f e (xs ++ concat xss)
=  {foldr 的给定性质}
 foldr f (foldr f e (concat xss)) xs
=  {使用 flip}
```

```
 flip (foldr f) xs (foldr f e (concat xss))
=  {归纳假设}
 flip (foldr f) xs (foldr (flip (foldr f)) e xss)
=  {foldr的定义}
 foldr (flip (foldr f)) e (xs:xss)
```

习题 H 答案 数学上表达为

```
sum (scanl (/) 1 [1..]) = e
```

因为 $\sum_{n=0}^{\infty} 1/n! = e$。从计算的角度讲，用有穷列表[1..n]代替[1..]给出 e 的一个逼近。例如：

```
ghci> sum (scanl (/) 1 [1..20])
2.7182818284590455
ghci> exp 1
2.718281828459045
```

标准引导库函数 exp 的输入为一个数 x，返回值为 e^x。此外，引导库函数 log 的输入为一个数 x，返回值为 $\log_e x$。如果需要其他底的对数，使用函数 logBase，其类型为

```
logBase :: Floating a => a -> a -> a
```

习题 I 答案 分情况合成一个更高效的定义。基本情况是

141

```
scanr f e [] = [e]
```

归纳情况 x:xs 是

```
 scanr f e (x:xs)
=  {根据说明}
 map (foldr f e) (tails (x:xs))
=  {tails.2}
 map (foldr f e) ((x:xs):tails xs)
=  {map的定义}
 foldr f e (x:xs):map (foldr f e) (tails xs)
=  {foldr.2以及说明}
 f x (foldr f e xs):scan f e xs
=  {要求:foldr f e xs = head (scanr f e xs)}
 f x (head ys):ys where ys = scanr f e xs
```

习题 J 答案 首先，有

```
subseqs = foldr op [[]]
 where op x xss = xss ++ map (x:) xss
```

利用融合定律得到

```
map sum . subseqs = foldr op [0]
 where op x xs = xs ++ map (x+) xs
```

再利用融合定律得到

```
maximum . map sum . subseqs = foldr op 0
 where op x y = y `max` (x+y)
```

这个定义效率不错。当然，sum . filter (>0)也可给出结果。

习题 K 答案

1. 表达式的值是

```
takePrefix nondec [1,3,7,6,8,9] = [1,3,7]
takePrefix (all even) [2,4,7,8] = [2,4]
```

恒等式为

```
takePrefix (all p) = takeWhile p
```

142

说明为

```
takePrefix p = last . filter p . inits
```

2. 恒等式为

```
none . f     = none
map f . none = none
map f . one  = one . f
```

3. 恒等式为

```
fst . fork (f,g) = f
snd . fork (f,g) = g
fork (f,g) . h   = fork (f.h,g.h)
```

4. 等式为

```
test p (f,g) . h = test (p.h) (f . h, g . h)
h . test p (f,g) = test p (h . f, h . g)
```

推理如下：

```
  map fst . filter snd . map (fork (id,p))
=  {filter 的定义}
  map fst . concat . map (test snd (one,none)) .
  map (fork (id,p))
=  {因为 map f . concat = concat . map (map f)}
  concat . map (map fst . test snd (one,none) .
  fork (id,p))
=  {test 的第二条定律; one 和 none 的定律}
  concat . map (test snd (one . fst,none) .
  fork (id,p))
=  {test 的第一条定律; fork 的定律}
  concat . map (test p (one . id, none . fork (id,p)))
=  {id 和 none 的定律}
  concat . map (test p (one,none))
=  {filter 的定义}
  filter p
```

143

5. 等式为

```
map (fork (f,g))  = uncurry zip . fork (map f,map g)
```

6. 推理如下：

```
  filter (p . foldl f e) . inits
=  {filter 的导出定律}
```

```
  map fst . filter snd .
  map (fork (id, p . foldl f e)) . inits
=  {zip的定律}
  map fst . filter snd . uncurry zip .
  fork (id, map (p . foldl f e)) . inits
=  {fork的定律}
  map fst . filter snd . uncurry zip .
  fork (inits, map (p . foldl f e) . inits)
=  {scan引理}
  map fst . filter snd . uncurry zip .
  fork (inits, map p . scanl f e)
```

因此

```
takePrefix (p.foldl f e)
   =  fst . last . filter snd . uncurry zip .
      fork (inits,map p . scanl f e)
```

6.9 注记

Gofer 是较早由 Mark Jones 设计的一个 Haskell 版本，命名来自于 GOod For Equational Reasoning。HUGS（The Haskell Users Gofer System）是 GHCi 的早期版本，也是本书第 2 版使用的系统，但是目前系统已不再维护。

许多人对于理解函数程序中的定律做出了贡献，不好一一列出。下列 Haskellwiki 网页含等式推理的例子和有关的讨论链接：

haskell. org/haskellwiki/Equational_reasoning_examples

关于最大段和问题的有趣历史的讨论，参考 Bentley 编写的《Programming Pearls》
144 (second edition)（Addison-Wesley，2000）。

效　率

效率是一直潜伏在最近讨论中的问题，现在是让这个重要问题浮上水面的时候了。获得效率的最好方法当然是找到解决问题的好算法。算法设计是一个很广阔的领域，不是本书的基本目的。不过，将在今后接触一些基本概念。本章集中在一个更基本的问题上：函数程序允许人们构造优美的表达式和定义，但是，人们是否知道对其求值的代价呢？美国的一位计算机科学家 Alan Perlis 曾经篡改奥斯卡·王尔德关于愤世嫉俗的人的名言：函数式程序设计员知道所有东西的价值，但是不知道它们的价格。

7.1　惰性求值

如第 2 章所讲，使用惰性求值时，诸如下面的表达式：

```
sqr (sqr (3+4))
```

其中 `sqr x = x*x`，被由外向里化简为最简单的形式。这表示函数 `sqr` 的定义首先被调用，其参数在需要时才被求值。下列求值序列遵循以上规则，但并不是惰性求值：

```
  sqr (sqr (3+4))
= sqr (3+4) * sqr (3+4)
= ((3+4)*(3+4)) * ((3+4)*(3+4))
= ...
= 2401
```


倒数第 2 行省略号隐藏了 `3 +4` 和 `7*7` 不少于 4 次的求值。显然，简单地将参数表达式代入函数定义表达式是一个非常低效的化简方法。

但是，惰性求值保证，如果需要参数的值，那么对参数的求值只进行一次。使用惰性求值时，化简序列大致如下：

```
  sqr (sqr (3+4))
= let x = sqr (3+4) in x*x
= let y = 3+4 in
  let x = y*y in x*x
= let y = 7 in
  let x = y*y in x*x
= let x = 49 in x*x
= 2401
```

对表达式 `3 +4` 的求值只有一次（`7*7` 的求值也只有一次）。这里使用了 `let` 将名称 `x` 和 `y` 约束为表达式，但是在 Haskell 实现中这些名称是指向表达式的指针。当一个表达式被化简为一个值时，该指针指向这个值，该值便可以被*共享*（shared）。

即便如此，“使用惰性求值时，参数只有在需要时被求值，而且只求值一次!”，这个标题并没有导出所有真相。考虑 `sqr (head xs)` 的求值。为了对 `sqr` 求值，必须对其参数求值，但是对 `head xs` 求值并不需要对 `xs` 的所有元素求值，只需计算到形如 `y:ys` 的

表达式即可。此时，head xs 返回 y，sqr (head xs)返回 y*y。更一般地，如果一个表达式是函数或者是数据构造函数（如(:)）应用于它的参数形式，则称表达式是首范式(head normal form)。每个范式（完全化简的式子）是首范式，但反过来不一定。例如，(e1,e2)是首范式（因为它等价于(,) e1 e2，其中(,)是二元组的构造函数），但是，只有当 e1 和 e2 均为范式时，该表达式才是范式。当然，对于数值和布尔值，范式和首范式没有区别。

"使用惰性求值时，参数只有在需要时被求值，并且只求值一次，而且此时只计算到首范式"，这种叙述虽然不如前面的叙述容易记忆，但是它的确是惰性求值更好的描述。

146 下面考虑函数 subseqs 归纳情况的两个定义，该函数返回一个列表的所有子序列：

```
subseqs (x:xs) = subseqs xs ++ map (x:) (subseqs xs)
subseqs (x:xs) = xss ++ map (x:) xss
                 where xss = subseqs xs
```

在第一个定义中，表达式 subseqs xs 在等式右边出现了两次，所以，当求一个列表的子序列时，它被求值两次。在第二个定义中，程序员察觉到这种重复工作，并用 where 子句确保 subseqs xs 只被求值一次（也可以使用 let 表达式）。

重要的是，作为程序员，可以控制使用哪个定义。让 Haskell 识别表达式的重复出现，并使用与内部 let 表达式等价的式子将其代替，这种方法是可行的。这就是众所周知的*公共子表达式消去*（common subexpression elemination）技术。但是，Haskell 没有这样做，而且有其道理：这样做可能引起*空间泄漏*（space leak）。subseqs (x:xs)第二个定义存在的问题：列表 subseqs xs 只构造一次，但是因为它的值被再次使用，即在第二个表达式 map (x:) xss 中，所以它在内存一直保存着。

对比以上两个定义：第一个定义花的时间长，因为其中有重复的计算；第二个定义更快（尽管仍然是指数阶的），但是可能很快会耗尽内存。这是因为，长度为 n 的列表有 2^n 个子序列。在程序设计中永远不能避开一分为二的原则：要避免重复的工作，必须用空间将一次计算的结果存储起来。

下面是一个相关的例子。考虑一个脚本中的下列两个定义：

```
foo1 n = sum (take n primes)
 where
 primes     = [x | x <- [2..], divisors x == [x]]
 divisors x = [d | d <- [2..x], x `mod` d == 0]


foo2 n = sum (take n primes)
primes     = [x | x <- [2..], divisors x == [x]]
divisors x = [d | d <- [2..x], x `mod` d == 0]
```

编写 foo1 的程序员将 primes 和 divisors 的定义局限于 foo1 的定义，因为预计这两个定义不会在脚本的其他定义中用到。编写 foo2 的程序员将这两个辅助函数的定义
147 作为全局定义或者*顶层*（top-level）定义。读者或许认为两种定义的效率没有区别，但是考虑下列与 GHCi 的交互。（命令:set +s 开启一些统计功能，在表达式求值后打印这些数据。）

```
ghci> :set +s
ghci> foo1 1000
3682913
(4.52 secs, 648420808 bytes)
ghci> foo1 1000
3682913
(4.52 secs, 648412468 bytes)
ghci> foo2 1000
3682913
(4.51 secs, 647565772 bytes)
ghci> foo2 1000
3682913
(0.02 secs, 1616096 bytes)
```

为什么 foo2 1000 的第二次求值比第一次快得多，但是 foo1 1000 的两次求值花的时间一样多？答案是 foo2 的定义需要列表 primes 的前 1000 个元素，所以求值后 primes 指向一个存储了前 1000 个素数的列表，第二次 foo2 1000 的求值不需要再次计算这些素数。在系统内部，脚本运行的空间已经增长，因为 primes 至少占据了 1000 个单位的内存。

第三个程序员选择如下定义 foo：

```
foo3 = \n -> sum (take n primes)
 where
 primes     = [x | x <- [2..], divisors x == [x]]
 divisors x = [d | d <- [2..x], x `mod` d == 0]
```

这里使用了兰姆达表达式在函数层表示 foo3，除表示方法外实质上与 foo1 完全一样。下列定义同样可行：

```
foo3 = sum . flip take primes
```

但是稍显模糊。现在可以进行求值：

```
ghci> foo3 1000
3682913
(3.49 secs, 501381112 bytes)
ghci> foo3 1000
3682913
(0.02 secs, 1612136 bytes)
```

148

同样，第二次求值比第一次快得多。这是为什么呢？

为了看清问题，可以将两个函数重写成下面形式：

```
foo1 n = let primes = ... in
         sum (take n primes)
foo3   = let primes = ... in
         \n -> sum (take n primes)
```

现在可以看出，每次对 foo1 1000 求值都需要对 primes 重新求值，因为 primes 是绑定在函数 foo1 的应用中，而不是绑定在该函数本身。理论上可以使第一个定义中的局部函数定义依赖于 n，这样的局部定义对于每个 n 均需要重新求值。在第二个定义中，局部函数定义绑定于函数本身（而且不可能依赖于函数的任何参数），结果是它们只需求值一次。当然，对 foo3 1000 求值后，primes 的局部定义将扩展成 1000 个元素的显式列表，接着是如何继续求值的方法。

7.2 空间的控制

假设将 sum 定义为 sum = foldl (+) 0。使用惰性求值策略，表达式 sum [1..1000] 的化简过程如下：

```
  sum [1..1000]
= foldl (+) 0 [1..1000]
= foldl (+) (0+1) [2..1000]
= foldl (+) ((0+1)+2) [3..1000]
= ...
= foldl (+) (..((0+1)+2)+ ... +1000) []
= (..((0+1)+2)+ ... +1000)
= ...
= 500500
```

求值过程中首先需要 1000 个单位空间来构造前 1000 个数之和的算术表达式，然后再对其求值。

149 更好的方法是混合使用惰性和勤奋求值：

```
  sum [1..1000]
= foldl (+) 0 [1..1000]
= foldl (+) (0+1) [2..1000]
= foldl (+) 1 [2..1000]
= foldl (+) (1+2) [3..1000]
= foldl (+) 3 [3..1000]
= ...
= foldl (+) 500500 []
= 500500
```

这里列表表达式[1..1000]的求值是惰性的，但是 foldl 的第二个参数累加和求值是勤奋的。交叉使用惰性求值和勤奋求值的结果是，求值过程使用的内存空间是常数。

上例表示，控制化简的次序将是有益的。提供这种方法的原始函数是 seq，类型为

```
seq :: a -> b -> b
```

对 x `seq` y 求值的次序是，先对 x 求值（到首范式），然后返回对 y 求值的结果。如果对 x 的求值不终止，x `seq` y 的求值也不终止。在 Haskell 中不能定义 seq，因此 Haskell 将该函数定义为原始函数。

现在考虑 foldl 的下列版本 foldl′，该函数对第二个参数是严格的：

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f e []     = e
foldl' f e (x:xs) = y `seq` foldl' f y xs
                    where y = f e x
```

Haskell 在标准引导库中提供了该函数（而且使用了这个乏味的名）。现在可以定义 sum = foldl′(+) 0，结果是求值使用常数空间。事实上，sum 是一个引导库函数，而且基本上是这样定义的。

是不是函数 foldl 现在变得多余，可以被 foldl′替代呢？在实际中是的，但是在理论上不是。可以构造 f、e 和 xs 使得

```
foldl f e xs ≠ foldl' f e xs
```

但是，如果 f 是严格的（如果$f \perp = \perp$，则称f是严格的），则以上两个表达式返回相同的结果。具体细节将在习题中讨论。 150

取平均值

有了以上准备工作，现在考虑一个很能说明问题的例子：如何计算一个数值列表的平均值（mean）。当然问题很简单，读者可能会想，只需将列表元素之和除以列表长度即可：

```
mean :: [Float] -> Float
mean xs = sum xs / length xs
```

这个定义有许多错误：不单单是右边的式子不是类型正确的！函数 length 在 Haskell 的类型是[a] -> Int，不进行类型显式转换不可以做 Float 和 Int 的除法。

在标准引导库中存在完成这种转换的函数：

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

注意到在第 3 章有两个转换函数：

```
toInteger   :: (Integral a) => a -> Integer
fromInteger :: (Num a) => Integer -> a
```

第一个将任何整型转换为整数，第二个将整数转换为一个数值。它们的复合将一个整型数，如 Int，转换为更一般的数，如 Float。

现在可以重写 mean 如下：

```
mean :: [Float] -> Float
mean xs = sum xs / fromIntegral (length xs)
```

这个定义的第二个问题是，定义默默地忽略了空列表的情况。如何处理 0/0？或者用错误信息的方式说明这种失败的情况，或者采用惯例说明空列表的均值是 0：

```
mean [] = 0
mean xs = sum xs / fromIntegral (length xs)
```

现在可以仔细检查 mean 真正有什么问题：它会引起空间泄漏。计算 mean [1..1000] 将会使得列表扩展，并在求和后继续保留在内存，因为在计算其长度时有指向列表的第二个指针。 151

可以使用所谓的元组（tupling）策略优化，用列表的一次遍历代替两次遍历。对本例来说方法很简单，定义函数 sumlen：

```
sumlen :: [Float] -> (Float,Int)
sumlem xs = (sum xs,length xs)
```

然后给出避免两次遍历的另一种定义。其中的计算容易实现，这里只给出结果：

```
sumlen []     = (0,0)
sumlen (x:xs) = (s+x,n+1)  where (s,n) = sumlen xs
```

函数 sumlen 定义的模式应该是熟悉的。另一种定义是

```
sumlen = foldr f (0,0)  where f x (s,n) = (s+x,n+1)
```

甚至更好的方法是用 foldl g 代替 foldr f，其中：

```
g (s,n) x = (s+x,n+1)
```

这样做的理由是第6章的定律：

```
foldr f e xs = foldl g e xs
```

对于所有列表 xs 成立，只要满足条件：

```
f x (g y z) = g (f x y) z
f x e = g e x
```

这两个条件的验证留作练习。

这也表明，可以用 foldl'定义：

```
sumlen = foldl' g (0,0)  where g (s,n) x = (s+x,n+1)
```

现在可以用下列定义代替备受批评的 mean 定义：

```
mean [] = 0
mean xs = s / fromIntegral n
          where (s,n) = sumlen xs
```

现在确实达到了使用常数空间计算 mean 的目的了吗？很不幸，没有。问题在
152 sumlen中，而且不容易发现。将定义稍稍展开，可以发现

```
foldl' f (s,n) (x:xs) = y `seq` foldl' f y xs
                        where y = (s+x,n+1)
```

哇，但是 y `seq` z 将 y 化简成首范式，而且(s + x,n = 1)已经是首范式。两个分量在这个计算完成前不会得到求值。这表示需要进一步深入 seq，并将 sumlen 重写如下：

```
sumlen = foldl' f (0,0)
  where f (s,n) x = s `seq` n `seq` (s+x,n+1)
```

最后，一切圆满，均值的计算在常数空间完成。

另外两个函数应用运算

函数应用是唯一不使用显式符号表示的运算。但是，Haskell 提供另外两个应用运算符，即($)和($!)：

```
infixr 0 $,$!
($),($!) :: (a -> b) -> a -> b
f $ x  = f x
f $! x = x `seq` f x
```

函数应用 f x 和 f $! x 的唯一区别是，在第二个表达式中，将 f 应用于参数 x 之前先计算 x 的值。函数应用 f x 和 f $ x 的唯一区别是($)（还有($!)）被说明具有最低的优先级0，而且是右结合的。这也恰恰是前面第一行优先级提供的声明。为什么需要这些运算呢？答案是，现在可以编写形如下面的式子：

```
process1 $ process2 $ process3 input
```

否则，需要如下表达：

```
process1 (process2 (process3 x))
(process1 . process2 . process3) x
```

不可否认的是($)在某些场合是很有用的，特别是在提交 GHCi 计算表达式时，所以

该运算符值得一提。根据以上讨论，严格应用运算符（$!）也是很有用的。 153

7.3 运行时间的控制

可以看出，在仪表盘上设置一个“勤奋”按钮是控制计算占用空间的一种最简单方法，但是如何控制运行时间呢？不幸的是，没有类似的能够加速计算的按钮，不过我们必须理解哪些会无意中降低计算速度。Haskell 平台编译器 GHC 附带文档中包含如何使得程序运行更快的有用建议。文档中有 3 个关键建议：

- 使用 GHC 的性能分析（profiling）工具。分析程序的运行时间和使用的空间是不可替代的。本书不讨论性能分析工具，但是有必要说明这些工具的存在。
- 改进程序性能的最好方法是使用更好的算法。本章开篇就提到这点。
- 使用库函数要比使用用户自己的函数好得多，因为这些库函数是由其他人认真设计和细心测试过的。用户可以设计一个比库 `Data.List` 提供的排序（`sort`）更好的排序函数，但是这会花费比写下 `import Data.List (sort)` 长得多的时间。使用 GHCi 时情况更是如此，因为 GHCi 会调用标准库中函数的编译版本。编译版本通常比解释版本快一个数量级。

GHC 文档提供的建议细节超出了本书范围，但是这里可以解释两个窍门。第一个，惰性求值的管理比勤奋求值的管理需要更多的开销，所以，如果知道一个函数的值会被用到，那么最好按下勤奋键。如文档所讲：“严格函数是你的好朋友”。

第二个建议有关类型。首先，`Int` 上的算术运算快于 `Integer` 上的算术，因为 Haskell 在处理潜在的大数时需要完成更多的工作。所以，在确保安全的条件下使用 `Int`，不使用 `Integer`。其次，如果将函数类型说明为所需要的具体类型，Haskell 便可以减少开支。例如，考虑 7.1 节定义的函数 `foo1` 的类型，在定义函数时没有说明其类型（实际上其他相关函数也没有说明类型），这是错误的，可以看出它的类型是

```
foo1 :: Integral a => Int -> a
```
154

如果真正感兴趣的是前 n 个素数之和，最好声明 `foo1` 的类型。例如：

```
foo1 :: Int -> Integer
```

有了这个更具体的定义，Haskell 无需随身携带类族 `Integral` 的方法和实例字典，因此减轻了负担。

这些建议可以减去常量的运行时间，但是不会影响渐进时间复杂度，即时间复杂度函数的阶。但是，人们有时会无意写出比预期渐进复杂度差的函数。考虑第 5 章讨论的笛卡儿积函数 `cp`：

```
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

定义看似优雅、清晰，但是与下列定义对比一下：

```
cp' = foldr op [[]]
 where op xs yss = [x:ys | x <- xs, ys <- yss]
```

第一个定义使用了直接递归，而第二个定义使用 `foldr` 来表达递归。两个“算法”是相同的，是不是？不过，有

```
ghci> sum $ map sum $ cp [[1..10] | j <- [1..6]]
33000000
(12.11 secs, 815874256 bytes)
ghci> sum $ map sum $ cp' [[1..10] | j <- [1..6]]
33000000
(4.54 secs, 369640332 bytes)
```

表达式 sum $ map sum 主要是为了强迫笛卡儿积进行完全求值。为什么第一个计算的速度只有第二个计算的1/3？答案是，检查第一个定义中消去列表概括的表示：

```
cp [] = [[]]
cp (xs:xss) = concat (map f xs)
    where f x = [x:ys | ys <- cp xss]
```

155 可以看出，每次 f 应用于 xs 的元素时，cp xss 都要求值一次。也就是，cp 在第一个例子中的计算次数远远多于它在第二个例子中的计算次数。目前难以给出更精确的数字，但是下面会引入估算运行时间的演算。不过，问题已经很清楚：cp 的简单递归定义无意导致了计算量大于预期。

更有效的笛卡儿积定义如下：

```
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
              where yss = cp xss
```

该定义的性能与使用 foldr 的定义性能相同。这里得到的经验是，看似简单的列表概括可能掩盖了有些表达式尽管只写了一次，但是会计算多次的事实。

7.4 时间分析

给定一个函数 f 的定义，用 $T(f)(n)$ 表示将 f 应用于“规模”为 n 的参数时，求值过程中化简步数在最坏情况下的渐进估计。为了稍后解释的原因，假定 T 的定义中使用的求值方法是勤奋求值，而不是惰性求值。

对于 T 的定义需要一些澄清。第一，$T(f)$ 表示 f 的一个给定定义的复杂度。时间复杂度是一个表达式的性质，而不是表达式值的性质。

第二，化简步数不完全等同于提交表达式求值和求得结果期间逝去的时间。对于较长且复杂的表达式化简，定义并没有考虑寻找下一个要化简的子表达式所需要的时间。基于这个原因，GHCi 的统计工具没有计算化简步数，只是输出逝去的时间。

第三，没有给规模的概念下定义，因为不同的场合有不同的规模度量。例如，对 xs ++ys求值时，最好用两个列表的长度二元组（m，n）表示规模。对 concat xss 求值时，可以用 concat xss 的长度作为规模，而当 xss 是长度为 m 的列表，其中每个元素又都是长度为 n 的列表时，用（m，n）作为规模可能更合理。

156 第四，也是最重要的一点，$T(f)(n)$ 是在勤奋求值模型下估算出来的。原因很简单，计算惰性求值中的化简步数很困难。例如，考虑定义 minimum = head . sort。在勤奋求值模型下，使用该定义对 n 个元素列表求最小值的时间复杂度是

$$T(\texttt{minimum})(n) = T(\texttt{sort})(n) + T(\texttt{head})(n)$$

换句话说，必须将长度为 n 的列表完全排序，然后取排序结果的第一个元素（应该是常数

时间的运算)。这个等式对于惰性求值不成立，因为找到 sort xs 的第一个元素所需要的化简步数只需要将 sort xs 化简为首范式即可。这个过程需要多少步依赖于 sort 所使用的算法。在勤奋求值模型下，时间的分析要简单得多，因为分析是可组合的（compositional)。因为惰性求值所需步数总是少于勤奋求值步数，所以 $T(f)(n)$ 的任何上界也是惰性求值步数的上界，而且在许多场合，一个下界也是惰性求值的下界。

为了举例分析时间复杂度，需要先介绍一点有关阶的表示。在讨论效率时一直在使用“步数正比于”这样的词语。现在是引入更简洁记法的时候了。给定两个自然数上的函数 f 和 g，如果存在大于 0 的常数 C_1 和 C_2 以及一个自然数 n_0 使得对于所有 $n > n_0$，$C_1 g(n) \leqslant f(n) \leqslant C_2 g(n)$ 成立，则称 f 是 g 阶的，记作 $f = \Theta(g)$。换句话说，对充分大的 n，f 的下界和上界都是 g 的常数倍。

这种记号被滥用到人们通常写得很简洁，如写成 $f(n) = \Theta(n^2)$，而不是写成正确的形式 $f(n) = \Theta(\lambda n.\, n^2)$。类似地，人们写成 $f(n) = \Theta(n)$，而不写 $f(n) = \Theta(id)$。使用 Θ 记号主要是隐藏常数，例如，可以写

$$\sum_{j=1}^{n} j = \Theta(n^2) \text{ 和 } \sum_{j=1}^{n} j^2 = \Theta(n^3)$$

在这里人们不在意忽略的常数。

有了这些背景知识，下面给出 3 个例子，说明如何分析一个计算的运行时间。首先考虑 concat 的两个定义：

```
concat xss  = foldr (++) [] xss
concat' xss = foldl (++) [] xss
```

如果 xss 是有穷列表，那么这两个定义是等价的。假定 xss 是长度为 m 的列表，其 157
中每个元素都是长度为 n 的列表，那么第一个定义给出：

$$T(\texttt{concat})(m,n) = T(\texttt{foldr (++) []})(m,n)$$
$$T(\texttt{foldr(++)[]})(0,n) = \Theta(1)$$
$$T(\texttt{foldr(++)[]})(m+1,n) = T(\texttt{++})(n,mn) + T(\texttt{foldr(++)[]})(m,n)$$

其中估算式 $T(\texttt{++})(n, mn)$ 来自将长度为 n 的列表与长度为 mn 的列表串联的工作。因为 $T(\texttt{++})(n, m) = \Theta(n)$，由此得到

$$T(\texttt{foldr(++)[]})(m,n) = \sum_{k=0}^{m} \Theta(n) = \Theta(mn)$$

对于 concat 的第二个定义，有

$$T(\texttt{concat'})(m,n) = T(\texttt{foldl(++)})(0,m,n)$$
$$T(\texttt{foldl(++)})(k,0,n) = \Theta(1)$$
$$T(\texttt{foldl(++)})(k,m+1,n) = T(\texttt{++})(k,n) + T(\texttt{foldl(++)})(k+n,m,n)$$

其中附加参数 k 表示 foldl 的第二个参数累积列表的长度。由此得到

$$T(\texttt{foldl(++)})(k,m,n) = \sum_{j=0}^{m-1} \Theta(k+jn) = \Theta(k+m^2 n)$$

因此 $T(\texttt{concat'})(m, n) = \Theta(m^2 n)$。结论是第 6 章所预测的，在 concat 定义中使用 foldr 而不使用 foldl 可以得到渐进运行更快的程序。

第二个例子计算 7.1 节讨论的 subseqs 的两个程序的运行时间，该函数的两个可能

的定义是

```
subseqs (x:xs)  = subseqs xs ++ map (x:) (subseqs xs)
subseqs' (x:xs) = xss ++ map (x:) xss
                  where xss = subseqs' xs
```

需要记住：（ⅰ）如果 xs 的长度为 n，那么 subseqs xs 的长度是 2^n；（ⅱ）串联和应用 map (x:) 的时间是 $\Theta(2^n)$，两个运行时间分析给出：

$$T(\texttt{subseqs})(n+1) = 2T(\texttt{subseqs})(n) + \Theta(2^n)$$

158

$$T(\texttt{subseqs'})(n+1) = T(\texttt{subseqs'})(n) + \Theta(2^n)$$

另外有 $T(\texttt{subseqs})(0) = \Theta(1)$。下面仅给出两个解（可以用简单归纳法证明）：

$$T(\texttt{subseqs})(n) = \Theta(n2^n)$$

$$T(\texttt{subseqs'})(n) = \Theta(2^n)$$

可见，后者比前者在渐进时间复杂度上快一个对数因子。

第三个例子计算本节开始讨论的 cp 的两个程序的运行时间。第一个定义是

```
cp []     = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

再次假定 xss 是长度为 m 的列表，列表的每个元素又都是长度为 n 的列表。这样 cp xss的长度是 n^m。由此得到

$$T(\texttt{cp})(0,n) = \Theta(1)$$

$$T(\texttt{cp})(m+1,n) = nT(\texttt{cp})(m,n) + \Theta(n^m)$$

这是因为应用(x:)于每个子序列需要 $\Theta(n^m)$ 步。最后得到的解是

$$T(\texttt{cp})(m,n) = \Theta(mn^m)$$

另一方面，用 foldr 定义的 cp 给出：

$$T(\texttt{cp})(0,n) = \Theta(1)$$

$$T(\texttt{cp})(m+1,n) = T(\texttt{cp})(m,n) + \Theta(n^m)$$

其解为 $T(\texttt{cp})(m, n) = \Theta(n^m)$。因此，第二个定义渐进更快，同样快一个对数因子。

7.5 累积参数

有时可以通过给函数增加一个额外参数，称为*累积参数*（accumulating parameter），从而改进计算的运行时间。典型的例子是函数 reverse：

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

159 对于这个定义，$T(\texttt{reverse})(n) = \Theta(n^2)$。为了找到线性时间的程序，假如定义：

```
revcat :: [a] -> [a] -> [a]
revcat xs ys = reverse xs ++ ys
```

显然，reverse xs = revcat xs []，所以，如果能够得到 revcat 的高效定义，那么也可以得到 reverse 的高效定义。为此，下面计算 revcat 的递归定义。基本情况 revcat [] ys =ys 留作练习，归纳情况如下：

```
  revcat (x:xs) ys
=  {revcat的定义}
  reverse (x:xs) ++ ys
=  {reverse的定义}
  (reverse xs ++ [x]) ++ ys
=  {(++)满足结合率}
  reverse xs ++ ([x] ++ ys)
=  {(:)的定义}
  reverse xs ++ (x:ys)
=  {revcat的定义}
  revcat xs (x:ys)
```

因此，有

```
revcat [] ys     = ys
revcat (x:xs) ys = revcat xs (x:ys)
```

对于运行时间，$T(\texttt{revcat})(m, n) = \Theta(m)$。特别是

$$T(\texttt{reverse}(n) = T(\texttt{revcat}(n,0) = \Theta(n)$$

由此得到一个线性时间的列表求逆程序。

下面是另一个例子。函数 `length` 定义如下：

```
length :: [a] -> Int
length []     = 0
length (x:xs) = length xs + 1
```

有 $T(\texttt{length})(n) = \Theta(n)$，所以，计算另一个定义不会在运行时间上有所收获。不过，如下定义函数 `lenplus`：

```
lenplus :: [a] -> Int -> Int
lenplus xs n = length xs + n
```

160

如果对 `lenplus` 完全重复以上对 `revcat` 进行的计算过程，则可得到

```
lenplus [] n     = n
lenplus (x:xs) n = lenplus xs (1+n)
```

计算过程可行的原因是(+)像(++)一样也满足结合律。现在如下定义 `length`：

```
length xs = lenplus xs 0 = foldl (\n x -> 1+n) 0 xs
```

这个定义的优点是，通过 `foldl'`代替 `foldl`，一个列表的长度可以在常数空间算出来。确实，这也是 Haskell 引导库函数 `length` 的定义。

敏锐的读者可能已经注意到，没有必要进行上面的计算。事实上，两个例子都是第 6 章描述的一个定律的特例，即

```
foldr (<>) e xs = foldl (@) e xs
```

只要下列条件成立：

```
x <> (y @ z) = (x <> y) @ z
      x <> e = e @ x
```

这两个特例是

```
foldr (\x n -> n+1) 0 xs = foldl (\n x -> 1+n) 0 xs
foldr (\x xs -> xs++[x]) [] xs
                        = foldl (\xs x -> [x]++xs) [] xs
```

这两个等式验证的细节留作练习。

作为累积参数的最后一个例子，我们从列表转向树。考虑下列数据声明：

```
data GenTree a = Node a [GenTree a]
```

这个类型的元素由一个带标记的结点和一些子树的列表构成。这种树出现在用状态和转移描述的问题中。一个结点的标记说明当前状态，子树的数目表示在当前状态下可能的转移数目。每棵子树都有一个标记说明该转移后的新状态，其子树描述在新状态下可以进行的转移，等等。

161 下面是计算一棵树中标记列表的函数：

```
labels :: GenTree a -> [a]
labels (Node x ts) = x:concat (map labels ts)
```

方法很简单：计算每棵子树的标记，将这些结果连接在一起，并将树的标记放在最后列表的前面。

现在分析这个程序在树 t 上的运行时间。为简单起见，假定 t 是高度为 h 的完美（perfect）k-元树。也就是说，如果 $h=1$，那么 t 没有子树；如果 $h>1$，那么 t 恰好有 k 棵子树，每棵子树的高度是 $h-1$。这种树上的标记数 $s(h, k)$ 满足

$$s(1,t) = 1$$
$$s(h+1,k) = 1 + ks(h,k)$$

由此得到解 $s(h, k)=\Theta(k^h)$。现在有

$$T(\texttt{labels})(1,k) = \Theta(1)$$
$$T(\texttt{labels})(h+1,k) = \Theta(1) + T(\texttt{concat})(k,s) + T(\texttt{map labels})(h,k)$$

其中 $s=s(h, k)$。表达式 $T(\texttt{map labels})(h, k)$ 估算将 `map labels` 应用于高度均为 h 的树构成的长度为 k 的列表的运行时间。一般地，给定一个长度为 k 的列表，其中列表元素的规模都是 n，那么

$$T(\texttt{map f})(k,n) = kT(\texttt{f})(n) + \Theta(k)$$

因为 $T(\texttt{concat})(k, s)=\Theta(ks)=\Theta(k^{h+1})$。所以，有

$$T(\texttt{labels})(h+1,k) = \Theta(k^{h+1}) + kT(\texttt{labels})(h,k)$$

这是因为 $\Theta(1)+\Theta(k)=\Theta(k)$。最后的解是

$$T(\texttt{labels})(h,k) = \Theta(hk^h) = \Theta(s \log s)$$

换句话说，利用以上定义计算树的标记的运行时间渐进地比树的规模大一个对数因子。

下面看一个累积参数能带来什么。定义 `labcat`：

```
labcat :: [GenTree a] -> [a] -> [a]
labcat ts xs = concat (map labels ts) ++ xs
```

除了添加一个列表参数 `xs` 外，将第一个参数由一棵树推广到树的列表。此时 `labels t = labcat [t] []`，所以，对 `labcat` 的任何改进将导致对 `labels` 的改进。

现在可以合成 labcat 的另一个定义。对于基本情况有

```
labcat [] xs = xs
```

162

对于归纳情况有如下推理：

```
  labcat (Node x us:vs) xs
=  {定义}
  concat (map labels (Node x us:vs)) ++ xs
=  {定义}
  labels (Node x us) ++ concat (map labels vs) ++ xs
=  {定义}
  x:concat (map labels us) ++ concat (map labels vs) ++ xs
=  {labcat的定义}
  x:concat (map labels us) ++ labcat vs xs
=  {也是labcat的定义}
  x:labcat us (labcat vs xs)
```

以上计算的结果是下列的 labels 程序：

```
labels t = labcat [t] []
labcat [] xs          = xs
labcat (Node x us:vs) = x:labcat us (labcat vs xs)
```

对于时间复杂度的分析，令 $T(\texttt{labcat})(h,\ k,\ n)$ 表示 labcat ts xs 的运行时间，其中 ts 是长度为 n 的树的列表，每棵树是高度为 h 的完美 k-元树（忽略了 xs 的规模，因为它不影响时间的估算）。那么

$$T(\texttt{labcat})(h,k,0) = \Theta(1)$$
$$T(\texttt{labcat})(1,k,n+1) = \Theta(1) + T(\texttt{labcat})(1,k,n))$$
$$T(\texttt{labcat})(h+1,k,n+1) = \Theta(1) + T(\texttt{labcat})(h,k,k) + T(\texttt{labcat})(h+1,k,n)$$

解前两个方程得到 $T(\texttt{labcat})(1,\ k,\ n) = \Theta(n)$，利用归纳法得到 $T(\texttt{labcat})(h,\ k,\ n) = \Theta(k^h n)$。因此

$$T(\texttt{labels})(h,k) = T(\texttt{labcat})(h,k,1) = \Theta(k^h) = \Theta(s)$$

这表示可以用正比于树的规模的时间计算一棵树的标记，相对第一个版本改进了一个对数因子。

163

7.6 元组

在讨论函数 mean 时提到了两个函数的元组。元组是累积参数的某种对偶：推广一个函数时给输出添加一个额外的结果，而不是添加额外的输入参数。

展示元组魅力的典型例子是斐波那契函数：

```
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

用这 3 个方程求 fib 的时间由下列式子估算：

$$T(\texttt{fib})(0) = \Theta(1)$$
$$T(\texttt{fib})(1) = \Theta(1)$$

$$T(\texttt{fib})(n) = T(\texttt{fib})(n-1) + T(\texttt{fib})(n-2) + \Theta(1)$$

因此，时间函数满足的方程非常类似于 fib 方程本身。事实上，$T(\texttt{fib})(n) = \Theta(\phi^n)$，其中 ϕ 是黄金比 $\phi = (1+\sqrt{5})/2$。这表示对 fib 在输入为 n 的求值运行时间是 n 的指数函数。

现在考虑如下定义的 fib2：

```
fib2 n = (fib n,fib (n+1))
```

显然 fib n = fst (fib2 n)。合成 fib2 的直接递归定义后得到

```
fib2 0 = (0,1)
fib2 n = (b,a+b)  where (a,b) = fib2 (n-1)
```

这个程序的运行时间是线性的。在这个例子中，元组的策略导致了戏剧性的改进，由指数时间降至线性时间。

利用一般定律刻画效率的改进是很有趣的。一个这样的定律有关下列计算：

```
(foldr f a xs, foldr g b xs)
```

如上式所示，两个 foldr 的应用涉及 xs 的两次遍历。所以，设计一个只遍历列表一次的定义不仅在时间上有收获，而且可能在空间性能上也有收获。事实上，有

164

```
(foldr f a xs, foldr g b xs) = foldr h (a,b) xs
```

其中：

```
h x (y,z) = (f x y,g x z)
```

结果可以用归纳法证明，将此留作习题。

作为另一个例子，再次由列表转向树。不过这次的树有些不同，它是标记叶的二叉树（leaf-labelled binary tree）：

```
data BinTree a = Leaf a | Fork (BinTree a) (BinTree a)
```

不同于以上讨论的 GenTree，一棵 BinTree 是一个有标记的叶，或者由两棵二叉子树构成的二叉树。

假如要用一个给定的标记列表构建一棵二叉树。更确切地说，我们想定义一个函数 build，而且对于任意非空列表 xs 满足

```
labels (build xs) = xs
```

其中 labels 返回一棵二叉树的标记：

```
labels :: BinTree a -> [a]
labels (Leaf x)   = [x]
labels (Fork u v) = labels u ++ labels v
```

我们感兴趣的是可能的优化，而且 labels 的定义表示可以用累积参数改进定义。尽管如此，但这并非主要目的，这种优化留作习题。

一种构建树的方法是将列表的一半元素用于构建左子树，另一半元素构建右子树：

```
build :: [a] -> BinTree a
build [x] = Leaf x
build xs  = Fork (build ys) (build zs)
            where (ys,zs) = halve xs
```

函数 halve 在 4.8 节出现过：

```
halve xs = (take m xs,drop m xs)
           where m = length xs `div` 2
```

可见，halve 将一个列表拆分为两个基本相等的列表。Halve 的定义涉及列表的遍历以便求出列表的长度，另外的两个（部分）遍历求出两个子列表。因此，应用元组方法是得到更好定义的首要选择。但是，对于 labels 暂时不考虑这个特定的优化，也不考虑 build 的定义满足其规格说明的证明。所以，已有 3 个计算留作习题，我们专注于第四个问题。 165

现在计算 build 的时间：

$$T(\texttt{build})(1) = \Theta(1)$$
$$T(\texttt{build})(n) = T(\texttt{build})(m) + T(\texttt{build})(n-m) + \Theta(n)$$
$$\textbf{where } m = n \text{ div } 2$$

将长度为 n 的列表拆分需要$\Theta(n)$ 步，然后分别用长度为 m 和长度为 $n-m$ 的列表递归地构建两棵子树。方程的解是

$$T(\texttt{build})(n) = \Theta(n \log n)$$

换句话说，使用以上方法构建一棵树更为耗时，其时间是列表长度的一个对数因子。

有了这个结果后，再来定义 build2：

```
build2 :: Int -> [a] -> (BinTree a,[a])
build2 n xs = (build (take n xs),drop n xs)
```

这个定义用列表的前 n 个元素构建一棵树，同时把剩余的列表作为结果返回。此时有

```
build xs = fst (build2 (length xs) xs)
```

所以，原函数可以用元组定义的函数表示。

现在的目的是构造 build2 的直接递归定义。首先，显然有

```
build2 1 xs = (Leaf (head xs),tail xs)
```

对于递归情况，由下式开始：

```
build2 n xs = (Fork (build (take m (take n xs)))
                    (build (drop m (take n xs))),
               drop n xs)  where m = n `div` 2
```

该等式是将 build 代入递归步骤得来的。这也表示下一步需要使用 take 和 drop 的性质。它们是，如果 m <= n，那么

```
take m . take n = take m
drop m . take n = take (n-m) . drop m
```

由此得到 166

```
build2 n xs = (Fork (build (take m xs))
                    (build (take (n-m) (drop m xs))),
               drop n xs)  where m = n `div` 2
```

利用 build2 的定义，可以将上式重写如下：

```
build2 n xs = (Fork u v, drop n xs)
              where (u,xs')  = build2 m xs
                    (v,xs'') = build2 (n-m) xs'
                    m        = n `div` 2
```

但是，作为最后一步，注意到

```
xs'' = drop (n-m) xs'
     = drop (n-m) (drop m xs)
     = drop n xs
```

因此，现在可以再次将 build2 重写为

```
build2 1 xs = (Leaf (head xs),tail xs)
build2 n xs = (Fork u v, xs'')
              where (u,xs')  = build2 m xs
                    (v,xs'') = build2 (n-m) xs'
                    m        = n `div` 2
```

估算其运行时间得到

$$T(\texttt{build2})(1) = \Theta(1)$$

$$T(\texttt{build2})(n) = T(\texttt{build2})(m) + T(\texttt{build2})(n-m) + \Theta(1)$$

其解为 $T(\texttt{build2})(n) = \Theta(n)$。因此，利用 build2 作为辅助函数，build 的运行时间改进了一个对数因子。

7.7 排序

排序是一个内容广泛的话题，人们可以花上很多愉快的时光讨论不同的算法。Knuth 在他的系列专著《The Art of Computer Programming》第三卷中用了 400 页讨论这一话题。即便如此，在纯函数式的背景下，某些排序的结论仍然需要重新陈述。本节对两个排序算
167 法进行简短讨论，并时刻关注对算法的可能优化。

归并排序

归并排序（Mergesort）在 4.8 节有定义：

```
sort :: (Ord a) => [a] -> [a]
sort []  = []
sort [x] = [x]
sort xs  = merge (sort ys) (sort zs)
           where (ys,zs) = halve xs
halve xs = (take m xs,drop m xs)
           where m = length xs `div` 2
```

实际上通过归并的排序有多种版本，标准引导库的函数 sort 使用的定义不同于这里的定义。

如前所述，halve 的定义看似相当低效，因为它需要对参数进行多次遍历。一种改进方法是利用标准引导库函数 splitAt，其说明如下：

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

该函数的引导库定义使用了元组转换：

```
splitAt 0 xs     = ([],xs)
splitAt n []     = ([],[])
splitAt n (x:xs) = (x:ys,zs)
                   where (ys,zs) = splitAt (n-1) xs
```

很容易使用下面两个事实计算 halve 的定义：对于任意 $0 < n$ 有

```
take n (x:xs) = x:take (n-1) xs
drop n (x:xs) = drop (n-1) xs
```

现在可以定义：

```
halve xs = splitAt (length xs `div` 2) xs
```

当然，这里仍然有两次遍历。

另一种改进 sort 的方法是定义：

```
sort2 n xs = (sort (take n xs),drop n xs)
```

有 sort xs = fst (sort2 (length xs) xs)，所以原来的排序函数可以在通用的排序中抽取。通过与前面的 sort 几乎完全一样的计算可以得到 168

```
sort2 0 xs = ([],xs)
sort2 1 xs = ([head xs],tail xs)
sort2 n xs = (merge ys zs, xs'')
             where (ys,xs')  = sort2 m xs
                   (zs,xs'') = sort2 (n-m) xs'
                   m         = n `div` 2
```

利用这个定义，不需要计算长度，也没有对 xs 的多次遍历。

另一种优化 halve 的方法是意识到，如果必须手工将一个列表分成两部分，不会如上这样做。如果要求将一个列表分成两部分，一定会用如下方法将列表元素分成两堆：

```
halve []       = ([],[])
halve [x]      = ([x],[])
halve (x:y:xs) = (x:ys,y:zs)
                 where (ys,zs) = halve xs
```

当然，这个定义返回的结果不同于前一定义的结果，如果拆分后的结果要排序，那么拆分后两个列表的元素顺序没有关系，重要的是所有元素都在某一子列表中。

现在已经总共有 3 种改进 sort 性能的方法，但结果是没有一个方法对 sort 总的运行时间有很大的改进。或许改进了几个百分点，但是没有实质的改进。而且，如果使用 GHCi 做函数求值器，没有一种定义在性能上比得过库函数 sort，因为这个函数是用编译的形式提供给用户的，而且编译后的函数通常运行会快十倍。当然，我们总是可以用 GHC 编译函数。

快速排序

第二个排序算法是有名的*快速排序*（Quicksort）。这个排序只需要两行 Haskell 代码描述：

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort [y | y <- xs, y < x] ++ [x] ++
              sort [y | y <- xs, x <= y]
```

169

这个定义非常优美，也是 Haskell 表现力的证明。但是，优美也是有代价的：程序在空间开销方面可能非常低效。情况与前面 mean 的程序一样。

在讨论如何优化代码之前，先来计算 $T(\texttt{sort})$。假设要对长度为 $n+1$ 的列表排序。第一个列表概括可能返回任何长度 k 介于 0 和 n 之间的列表，因此第二个列表概括结果的长度是 $n-k$。因为运行时间函数是最后运行时间的估计，所以需要取这些可能性中最大者：

$$T(\texttt{sort})(n+1) = \max\,[\,T(\texttt{sort})(k) + T(\texttt{sort})(n-k) \mid k \leftarrow [0\,..\,n]\,] + \Theta(n)$$

其中 $\Theta(n)$ 项表示计算两个列表概括和完成列表串联的运行时间。顺便指出，以上列表概括是在数学表达式中的应用，而不是 Haskell 表达式中的应用。如果列表概括在程序设计中是有用的记号，那么它们在数学中一样是有用的。

最坏的情况在 $k=0$ 或者 $k=n$ 时发生，尽管不是很明显。因此，有

$$T(\texttt{sort})(0) = \Theta(1)$$

$$T(\texttt{sort})(n+1) = T(\texttt{sort})(n) + \Theta(n)$$

由此得到解 $T(\texttt{sort}) = \Theta(n^2)$。所以，快速排序在最坏情况下是平方阶的算法。这是算法本身的特点，无关 Haskell 的表达方式。快速排序的名望来自两个原因，但是两者在纯函数程序情况下都不成立。第一，当快速排序用数组实现，而不是用列表时，划分过程可以原地（in place）完成，无需更多额外空间。第二，在对输入的一些合理假设下，快速排序的平均情况性能是 $\Theta(n\log n)$，而且其中的常数很小。在函数式程序设计情况下，这个常数不是很小，而且有比快速排序更好的排序方法。

有了这些分析，让我们看看在不做实质性改变的情况（不是一个完全不同的排序算法）下如何改进算法。为了避免划分时做两次遍历，定义：

```
partition p xs = (filter p xs, filter (not . p) xs)
```

这是另一个将两个定义作为二元组以节约一次遍历的例子。因为 filter p 可以表示成
170 foldr 的一个特例，故可利用 foldr 的元组定律得到

```
partition p = foldr op ([],[])
 where op x (ys,zs) | p x       = (x:ys,zs)
                    | otherwise = (ys,x:zs)
```

现在可以定义：

```
sort []     = []
sort (x:xs) = sort ys ++ [x] ++ sort zs
              where (ys,zs) = partition (<x) xs
```

不过这个定义仍然有空间泄漏问题。为了解释原因，将递归情况写成如下等价形式：

```
sort (x:xs) = sort (fst p) ++ [x] ++ sort (snd p)
              where p = partition (<x) xs
```

假如 x:xs 的长度为 $n+1$，而且列表元素是严格递减的，那么 x 是列表的最大元素，p 是长度分别为 n 和 0 的列表二元组。显示第一个递归调用结果引发对 p 的求值，但是 p 的第一个分量占用的空间不能释放，因为在第二个递归调用中有指向 p 的引用。在第二个递归调用中间，有更多的列表二元组生成并保留在内存。总之，对长度为 $n+1$ 的严格递减列表进行排序需要的总空间是 $\Theta(n^2)$ 单位。这意味着现实中对于大输入的排序会因为没有足够空间而退出。

解决方法是强制 partition 的求值，同样重要的是，绑定 ys 和 zs 到二元组的分量，而不是 p 本身。

一种得到满意结果的方法是引入两个累积参数，如下定义 sortp：

```
sortp x xs us vs = sort (us ++ ys) ++ [x] ++
                   sort (vs ++ zs)
                   where (ys,zs) = partition (<x) xs
```

这样便有

```
sort (x:xs) = sortp x xs [] []
```

现在可以为 sortp 合成一个直接递归定义。基本情况定义为

```
sortp x [] us vs = sort us ++ [x] ++ sort vs
```

171

对于归纳情况 y:xs，假设 y < x。那么

```
  sortp x (y:xs) us vs
=   {sortp 的定义以及 (ys,zs) = partition (<x) xs}
  sort (us ++ y:ys) ++ [x] ++ sort (vs ++ zs)
=   {见以下的断言}
  sort (y:us ++ ys) ++ [x] ++ sort (vs ++ zs)
=   {sortp 的定义}
  sortp x (y:us) vs
```

其中的断言是，如果 as 是 bs 的一个置换，那么 sort as 和 sort bs 返回同一个结果。断言直观上是显然的：对一个列表排序，结果不依赖于输入元素的排列次序，只依赖于列表中的元素。形式证明略去。

对于 x <= y 的情况再进行类似的计算，并使 sortp 局部于 sort 的定义，得到最后的程序：

```
sort []     = []
sort (x:xs) = sortp xs [] []
  where
  sortp [] us vs     = sort us ++ [x] ++ sort vs
  sortp (y:xs) us vs = if y < x
                       then sortp xs (y:us) vs
                       else sortp xs us (y:vs)
```

尽管程序看上去没有以前那么漂亮，但是至少空间复杂度是$\Theta(n)$。

7.8　习题

习题 A　函数 sort 的一个简单定义是

```
sort []     = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys)
     = if x <= y then x:y:ys else y:insert x ys
```

这种排序方法称为*插入排序*（insertion sort）。请用惰性求值将 sort [3,4,2,1] 化为 172
首范式。然后回答下列问题：（ i ）将 head . sort 应用于长度为 n 的列表时，对该表达式求值需要多长时间（作为 n 的函数）？（ ii ）使用勤奋求值又需要多长时间？（ iii ）使用

惰性求值时，插入排序完成的比较序列是否与下列选择排序（selection sort）完成的比较序列一样？

```
sort [] = []
sort xs = y:sort ys  where (y,ys) = select xs

select [x]    = (x,[])
select (x:xs) | x <= y    = (x,y:ys)
              | otherwise = (y,x:ys)
              where (y,ys) = select xs
```

习题 B 请给出 length 的一个定义，其求值可用常数空间完成。写出 length 的第二个定义，其求值在常数空间完成，但是不使用原始运算 seq（直接地或者间接地）。

习题 C 请给出 f、e 和 xs 使得

$$\texttt{foldl f e xs} \neq \texttt{foldl' f e xs}$$

习题 D 如果如下定义 cp：

```
cp []       = [[]]
cp (xs:xss) = [x:ys | ys <- cp xss, x <- xs]
```

请问这样定义 cp 是不是像使用 foldr 定义的 cp 一样高效？是，不是，还是也许？下面做一个计算。使用 foldr 的融合律计算下列函数的高效定义：

```
fcp = filter nondec . cp
```

173 关于 nondec 的定义参见 4.7 节。

习题 E 假设对于 $n \geq 2$ 有下列递推式：

$$T(1) = \Theta(1)$$

$$T(n) = T(n \text{ div } 2) + T(n - n \text{ div } 2) + \Theta(n)$$

请证明 $T(2^k) = \Theta(2^k k)$。由此证明 $T(n) = \Theta(n\log n)$。

习题 F 证明：

```
foldr (\x n -> n+1) 0 xs = foldl (\n x -> 1+n) 0 xs
foldr (\x xs -> xs++[x]) [] xs
                         = foldl (\xs x -> [x]++xs) [] xs
```

习题 G 证明：如果 h x (y,z) = (f x y,g x z)，则对于所有有穷列表 xs 有

```
(foldr f a xs,foldr g b xs) = foldr h (a,b) xs
```

一个复杂的问题是：这个结果对于所有的列表 xs 成立吗？

再定义一个函数 h 使得

```
(foldl f a xs,foldl g b xs) = foldl h (a,b) xs
```

习题 H 回顾定义：

```
partition p xs = (filter p xs, filter (not . p) xs)
```

请将两个分量表示成 foldr 的两个特例。由此利用习题 G 的结果计算 partition 的另一个定义。

定义：

```
part p xs us vs = (filter p xs ++ us,
                   filter (not . p) xs ++ vs)
```

请计算 partition 的另一个定义，并使用 part 作为局部定义。

习题 I　回顾定义：

174

```
labels :: BinTree a -> [a]
labels (Leaf x)   = [x]
labels (Fork u v) = labels u ++ labels v
```

计算 T(labels)(n)，其中 n 是树中叶的个数。请利用累积参数技术给出快速计算 labels 的方法。

证明 labels (build xs) = xs 对于所有有穷非空列表成立。

习题 J　定义 select k = (!! k) . sort，其中 sort 是最初给出的快速排序。所以，select k 选择非空有穷列表中第 k 最小元素。第 0 最小元素是最小的元素，第 1 最小元素是下一个最小元素，等等。请给出 selcet 的一个更高效定义，并估算其运行时间。

7.9　答案

习题 A 答案

```
  sort [3,4,1,2]
= insert 3 (sort [4,1,2])
= ...
= insert 3 (insert 4 (insert 1 (insert 2 [])))
= insert 3 (insert 4 (insert 1 (2:[])))
= insert 3 (insert 4 (1:2:[]))
= insert 3 (1:insert 4 (2:[]))
= 1:insert 3 (insert 4 (2:[]))
```

将 head . sort 应用于长度为 n 的列表的求值需要$\Theta(n)$ 步。使用勤奋求值大约需要 n^2 步。对于（iii），答案是“是的”。你可能会想，我们定义了插入排序，但是，在惰性求值策略下它是选择排序。这里得到的经验是，在惰性求值策略下，事情并不总是我们所想象的那样。

习题 B 答案　对于第一部分，下面的定义满足要求：

```
length = foldl' (\n x -> n+1) 0
```

对于第二部分，一个解如下：

175

```
length          = length2 0
length2 n []    = n
length2 n (x:xs) = if n==0 then length2 1 xs
                   else length2 (n+1) xs
```

测试 n ==0 迫使对第一个参数求值。

习题 C 答案　定义 f n x = if x ==0 then undefined else 0，则有

```
foldl f 0 [0,2]  = 0
foldl' f 0 [0,2] = undefined
```

习题 D 答案　答案是，也许！尽管给出的 cp 定义是高效的，但是，它返回列表的列表元素的排序次序不同于书中其他任何定义的结果。如果只关心结果的集合，那么次序关系不大，但是对那些要查找一个满足一定性质的列表的程序，该定义对程序的运行时间和

结果可能会有影响。

根据融合律，必须定义一个函数使得

```
filter nondec (f xs yss) = g xs (filter nondec yss)
```

其中 `f xs yss = [x:ys | x <- xs,ys <- yss]`。这样便有

```
filter nondec . cp
 = filter nondec . foldr f [[]]
 = foldr g [[]]
```

现在有

```
nondec (x:ys) = null ys || (x <= head ys && nondec ys)
```

由此得到

```
g xs [[]] = [[x] | x <- xs]
g xs yss  = [x:ys | x <- xs, ys <- yss, x <= head ys]
```

习题 E 答案 对于第一部分，有

176

$$T(2^k) = 2T(2^{k-1}) + \Theta(2^k)$$

根据归纳法，可以证明 $T(2^k) = \sum_{i=0}^{k} \Theta(2^k)$。归纳步骤如下：

$$\begin{aligned} T(2^k) &= 2\sum_{i=0}^{k-1} \Theta(2^{k-1}) + \Theta(2^k) \\ &= \sum_{i=0}^{k-1} \Theta(2^k) + \Theta(2^k) \\ &= \sum_{i=0}^{k} \Theta(2^k) \end{aligned}$$

因此，$T(n) = \Theta(k2^k)$。现在假定 $2^k \leqslant n < 2^{k+1}$，那么

$$\Theta(k2^k) = T(2^k) \leqslant T(n) \leqslant T(2^{k+1}) = \Theta((k+1)2^{k+1}) = \Theta(k2^k)$$

所以，$T(n) = \Theta(k2^k) = \Theta(n\log n)$。

习题 F 答案 定义 `x <> n = n+1` 以及 `n @ x = 1+n`，则有

```
(x <> n) @ y = 1+(n+1) = (1+n)+1 = x <> (n @ y)
```

第二个证明类似。

习题 G 答案 归纳步骤如下：

```
  (foldr f a (x:xs),foldr g b (x:xs)
= (f x (foldr f a xs),g x (foldr g b xs))
= h x (foldr f a xs,foldr g b xs)
= h x (foldr h (a,b) xs
= foldr h (a,b) (x:xs)
```

对于复杂问题的回答：不是。因为在 Haskell 中（⊥，⊥）和⊥是不同的值。例如，假设定义 `foo (x,y) = 1`，则有

```
foo undefined = undefined
foo (undefined,undefined) = 1
```

对于最后一部分，定义 `h` 如下：

```
h (y,z) x = (f y x,g z x)
```

习题 H 答案　我们有 `filter p = foldr (op p) []`，其中：　177

```
op p x xs = if p x then x:xs else xs
```

现在有

```
 (op p x ys,op (not . p) x zs)
= if p x then (x:ys,zs) else (ys,x:zs)
```

因此，有

```
partition p xs = foldr f ([],[]) xs
  where f x (ys,zs) = if p x
                      then (x:ys,zs)
                      else (ys,x:zs)
```

对最后一个问题，有

```
partition p xs = part p xs [] []
part p [] ys zs = (ys,zs)
part p (x:xs) ys zs = if p x
                      then part p xs (x:ys) zs
                      else part p xs ys (z:zs)
```

习题 I 答案　记着 T 表示最坏情况时间复杂度。对于 `labels` 的最坏情况是树的每棵右子树都是一个叶。因此，有

$$T(\texttt{labels})(n) = T(\texttt{labels})(n-1) + \Theta(n)$$

其中 $\Theta(n)$ 表示将长度为 $n-1$ 的列表与长度为 1 的列表串联的运行时间。因此，有

$$T(\texttt{labels})(n) = \sigma_{j=0}^{n}\Theta(j) = \Theta(n^2)$$

使用累积参数方法得到

```
labels t              = labels2 t []
labels2 (Leaf x) xs   = x:xs
labels2 (Fork u v) xs = labels2 u (labels2 v xs)
```

并且 $T(\texttt{labels2})(n) = \Theta(n)$。这个定义将 `labels` 的运行时间由二次方改进为线性。

对于证明 `labels (build xs) = xs`，其中的归纳假设为对于所有长度严格小于 `xs` 长度的列表该等式成立。归纳步骤如下：　178

```
  labels (build xs)
=  {设xs的长度至少为2
    并令(ys,zs) = halve xs}
  labels (Fork (build ys) (build zs))
=  {labels的定义}
  labels (build ys) ++ labels (build zs)
=  {归纳假设，因为ys和zs的长度严格小于xs的长度}
  ys ++ zs
=  {halve xs的定义}
  xs
```

这里使用了一般归纳（general induction）。要证明 P（xs）对于所有有穷列表 xs 成立，只要证明：（ⅰ）P([]) 成立；（ⅱ）在假设性质 P 对于所有长度严格小于 xs 长度列表成立的前提下，P(xs) 也成立。

习题 J 答案 一个关键性质是

```
(xs ++ [x] ++ ys)!!k | k < n   = xs!!k
                     | k==n    = x
                     | k > n   = ys!!(n-k)
                     where n = length xs
```

另一个关键性质是，对一个列表排序不改变列表的长度。因此，有

```
select k []     = error "list too short"
select k (x:xs) | k < n     = select k ys
                | k==n      = x
                | otherwise = select (n-k) zs
                where ys = [y | y <- xs, y < x]
                      zs = [z | z <- xs, x <= z]
                      n  = length ys
```

对于长度为 n 的列表，最坏情况运行时间是当 $k=0$ 并且 ys 的长度为 $n-1$ 时，即 x:xs是严格递减的。因此，有

$$T(\texttt{select})(0,n) = T(\texttt{select})(0,n-1) + \Theta(n)$$

其解为 $T(\texttt{select})(0, n) = \Theta(n^2)$。但是，假定排序结果的任何排列都等可能成为排序的输入，那么 $T(\texttt{select})(k, n) = \Theta(n)$。

7.10 注记

算法设计方面的书有很多，不过有两本书关注于函数式程序设计方面的算法设计：由 Fethi Rabbi 和 Guy Lapalme 编著的《A Functional Programming Approach》（second edition）（Addison-Wesley，1999），以及由我编著的《Pearls of Functional Algorithm Design》（Cambridge，2010）。

性能分析工具的信息包含在 Haskell 平台文档中。排序算法的参考资料见 Don Knuth 的《The Art of Computer Programming，Volume 3：Sorting and Searching》（second edition）（Addison-Wesley，1998）。

179 ≀ 180

第8章
Thinking Functionally with Haskell

精美打印

本章介绍在 Haskell 中如何构建一个小函数库的例子。库是由一些特定类型和函数构成的集合，它们可供用户完成某种任务。这里选择的任务是*精美打印*（pretty-printing），即将文本用多行显示，使得内容更容易阅读和理解。这里将忽略许多改进可阅读性的特性，如颜色或者字号，而只是将关注点放在每一行的换行位置以及缩进量上。该库不能用于数学内容的格式布置，但是可以帮助展示树状信息，或将词的列表显示为段落。

8.1 问题背景

首先考虑条件表达式的显示问题。本书使用了三种方式显示这种表达式：

```
if p then expr1 else expr2

if p then expr1
else expr2

if p
then expr1
else expr2
```

这三种格式分别占用一行、两行和三行，都是可以接受的，但是下列两种不可接受： 181

```
if p then
expr1 else expr2

if p
then expr1 else expr2
```

至于哪种格式可接受、哪种格式不可接受，这是由作者决定的。用户或许不同意作者的选择（有的用户同意），而且一个灵活的库应该允许用户自己规定可接受的格式。总之，有两个问题需要回答。第一，如何描述可接受的格式，拒绝不可接受的格式？第二，如何在可接受的格式中进行选择？

对于第二个问题的简单回答是，可以根据所允许的宽度做出选择。例如，用户可能选择占最少行的格式，条件是每行能够在给定的宽度内显示。后面会进一步讨论这个问题。

至于第一个问题，一种答案是写出所有可接受的格式。这需要大量的书写工作。一个更好的方法是给用户提供一种适当的*格式描述语言*（layout description language）。大致的想法如下：

```
if p <0> then expr1 (<0> + <1>) else expr2 +
if p <1> then expr1 <1> else expr2
```

其中 <0> 表示一个空格，<1> 表示换行，+ 表示“或者”。以上表达式可以生成上述看到的三种可接受的格式。但是，为用户提供这种不受约束的选择权的问题是，如果不进一步查看每个选择，就很难决定最好的格式，而查看每个选择需要花很多时间。

另一种方法是让用户只能使用一个函数库提供的某些函数和运算描述格式，因此只允许受限制的格式。例如，考虑下面的描述：

```
group (group (if p <1> then expr1) <> <1> else expr2)
```

其中 group 在一组格式基础上填加另一组格式，它将一组格式中的每个 <1> 用 <0> 替换，由此使得每个格式只占一行，(< >) 表示将串联提升到格式集合的连接。例如：

```
group (if p <1> then expr1)
  = {if p <0> then expr1, if p <1> then expr1}
```

182

```
group (if p <1> then expr1) <> <1> else expr2
  = {if p <0> then expr1 <1> else expr2,
     if p <1> then expr1 <1> else expr2}
group (group (if p <1> then expr1) <> <1> else expr2)
  = {if p <0> then expr1 <0> else expr2,
     if p <0> then expr1 <1> else expr2,
     if p <1> then expr1 <1> else expr2}
```

因此，使用两个 group 就可以表达以上三种可接受的格式。

显示条件表达式格式问题还有另一个方面需要考虑。如果 expr1 和 expr2 本身是条件表达式怎么办？或许可允许如下的格式：

```
if p
then if q
     then expr1
     else expr2
else expr3
```

关键是在描述语言中应该允许缩进（indentation）。缩进指每个换行后添加适当数目的空格。这个想法可以通过一个函数 nest 实现，使得 nest i x 表示在格式 x 中每个换行后添加 i 个空格。

8.2 文档

为了统一名词，把表示一段文本的可能格式集合的对象称作文档（document）。文档将作为稍后定义的类型 Doc 的元素。另一方面，一种格式仅仅是一个串：

```
type Layout = String
```

在这里对于文档到底是什么有意含糊其词，因为今后将给出 Doc 的两种定义。目前只集中考虑格式打印库应该提供文档上的哪些运算。

第一个运算是一个函数：

183

```
pretty :: Int -> Doc -> Layout
```

其参数是一个给定的行宽和一个文档，返回最好的格式。如何给这个函数一个高效的定义正是本章的主要内容。

第二个运算是函数：

```
layouts :: Doc -> [Layout]
```

它用一个列表返回一个文档的可能格式的集合。为什么有了函数 pretty 还需要这个函数呢？原因是，找出描述用户认为可接受的格式定义需要一些实验。实验的方法是先给出一

个初始定义，然后通过查看定义在一些例子上的结果格式再对定义进行修正。通过这种方法，可以看出应该排除哪些格式，应该添加哪些格式。所以，无论文档的最终表示如何，layouts 都将为用户提供一个格式诊断工具。

剩余的运算用于构建文档。首先是将两个文档串联成一个新文档的运算：

```
(<>) :: Doc -> Doc -> Doc
```

文档串联运算显然应该满足结合律，所以对任何文档 x、y 和 z，<>的实现应满足下面等式：

```
(x <> y) <> z) = x <> (y <> z)
```

每当有满足结合律的运算时，通常需要有一个单位元，所以需要提供一个空文档：

```
nil :: Doc
```

使得对于任意文档 x，nil <> x = x 和 x <> nil = x 成立。

下一个运算是函数：

```
text :: String -> Doc
```

它将一个不含换行符的串转换为一个文档。为了提供多行的文档，函数库提供另一个基本文档：

```
line :: Doc
```

例如：

```
text "Hello" <> line <> text "World!"
```

184

是由两行构成的单一格式文档。你可能觉得 line 在这里不必要，因为可以允许在文本串中加入换行符，但是，如果将一个文档缩进，就必须查看每个文本的内容。更好的方法是提供一个显式的换行文档，这样就可以选择在哪里换行了。

下一个运算是提供文档嵌套的函数：

```
nest :: Int -> Doc -> Doc
```

其中 nest i 在每个换行后面插入 i 个空格，从而实现文档缩进。需要强调的是，缩进不是在文档的开始位置完成的，除非文档以换行开始。后面将解释这种选择的原因。

最后，提供下列函数，从而完成一个含 8 个运算的格式打印库：

```
group :: Doc -> Doc
```

这是提供多种格式的函数。函数 group 在其参数文档上添加一个额外格式，该格式不含换行，仅由一个文本行构成。

前面已经给出 8 个命名的运算，并给出它们的非正式描述，但是，能否对它们满足的性质以及它们之间的关系给出更精确的描述呢？更基本的问题是，这些运算是否足够灵活，能否提供一组合理的格式。

首先考虑我们希望什么样的定律成立，找到这些定律能够增强我们对集成工具箱的信心，相信该工具箱是合理的、自然的集成，而且没有漏掉某个关键的工具。这些定律也会影响运算的含义，指导运算的实现。前面已经断定运算（<>）应该具有单位元 nil，而且满足结合律，但是还需要其他定律吗？

是的，对于 text 需要下面性质成立：

```
text (s ++ t) = text s <> text t
text ""       = nil
```

用数学语言来说，这个性质说明 text 是串的串联运算到文档串联运算的同态。对于这么简单的事情来讲，这是一个令人印象深刻（也许同时令人生畏）的名词。注意，串的
185 串联满足结合律意味着文档串联也满足结合律，至少对于 text 是这样的。

对于 nest，需要下面的等式成立：

```
nest i (x <> y)   = nest i x <> nest i y
nest i nil        = nil
nest i (text s)   = text s
nest i line       = line <> text (replicate i ' ')
nest i (nest j x) = nest (i+j) x
nest 0 x          = x
nest i (group x)  = group (nest i x)
```

所有这些等式（可能最后一个除外）都很合理，有些可以冠以数学的术语（nest i 对于串联可分配，nest 是数值加法到函数复合的同态，并且 nest i 与 group 可交换）。假如 nest 是在文档的开始缩进，则第三条定律不成立；如果允许文本串包含换行符，这条定律也不成立。最后一条定律成立的原因是分组添加了没有换行的格式，而且 nest 对这种格式没有任何影响。更详细的说明参见习题 D。

对于格式的性质，要求：

```
layouts (x <> y)   = layouts x <++> layouts y
layouts nil        = [""]
layouts (text s)   = [s]
layouts line       = ["\n"]
layouts (nest i x) = map (nestl i) (layouts x)
layouts (group x)  = layouts (flatten x) ++ layouts x
```

其中运算（<++>）是提升的串联：

```
xss <++> yss = [xs ++ ys | xs <- xss, ys <- yss]
```

函数 nestl :: Int -> Layout -> Layout 的定义如下：

```
nestl i    = concat . map (indent i)
indent i c = if c=='\n' then c:replicate i ' ' else [c]
```

最后，flatten :: Doc -> Doc 是将一个文档中的换行及其相关的缩进用一个空格代替后得到的单一格式文档。该函数不在文档库的公共接口中出现，但是在库内部需要用到该函数。完成代数定律的描述需要该函数，可以说在这个意义下该函数是不可缺失的工具。

186 函数 flatten（扁平化）需要满足下列条件：

```
flatten (x <> y)   = flatten x <> flatten y
flatten nil        = nil
flatten (text s)   = text s
flatten line       = text " "
flatten (nest i x) = flatten x
flatten (group x)  = flatten x
```

现在一共有 24 条定律（关于 <>1 条，nil 和 text 各 2 条，关于 nest 有 7 条，layouts 和 flatten 各 6 条）。许多定律看起来像包含 nil、text 等构造函数的数据类型上的 Haskell 函数定义。更多详情参加 8.6 节。

8 个运算似乎足矣，但是，这些运算是否足以灵活地描述用户想要的格式？布丁做得好不好，尝尝才知道。所以，我们马上要考虑 3 个例子。在此之前，不管是好是坏，需要

实现文档以便测试这些例子。

8.3 一种直接实现

文档表示的一种显然选择是将文档等同于它的格式列表：

```
type Doc = [Layout]
```

这种表示称为浅嵌入（shallow embedding）。对于浅嵌入，库函数直接用相关的值（这里是 layouts）实现。稍后我们将舍弃这种表示，选择一种更结构化的方法，但就目前来讲，这是显然的尝试。

下面是库函数的实现（pretty 的实现稍后再讲）：

```
layouts   = id
x <> y    = x <++> y
nil       = [""]
line      = ["\n"]
text s    = [s]
nest i    = map (nestl i)
group x   = flatten x ++ x
flatten x = [flattenl (head x)]
```

我们已经定义了 nestl，flattenl 的定义如下： 187

```
flattenl :: Layout -> Layout
flattenl [] = []
flattenl (c:cs)
   | c=='\n'   = ' ':flattenl (dropWhile (== ' ') cs)
   | otherwise = c:flattenl cs
```

这个实现满足前面的 24 条定律吗？我们来挨个检查一遍。提升的串联 <++> 具有单位元[[]]，并满足结合律，所以前 3 条定律成立。容易验证关于 text 的两条定律，关于 layouts 的 6 条定律也显然成立。关于 nest 的定律除两条外都成立。这两条例外是

```
nest i . nest j = nest (i+j)
nest i . group  = group . nest i
```

验证它们需要下一点功夫（见习题 C 和习题 D）。剩下的是 flatten，其中 3 条容易说明，并且稍花点时间就可以证明（见习题 E 和习题 F）：

```
flatten . nest i = flatten
flatten . group  = flatten
```

但困难的是下面的定律：

```
flatten (x <> y) = flatten x <> flatten y
```

这条定律不成立。如果取 x = line，y = text "hello"，那么

```
flatten (x <> y) = ["hello"]
flatten x <> flatten y = ["   hello"]
```

可见这两个结果不同。其原因是 flatten 去除了嵌套的影响，但是如果在非嵌套文档中换行后有空格，flatten 并没有删除这些空格。另一方面，flattenl 删除文档中每个换行后的空格。

我们不准备修正这个缺陷，而是接受这个不太完美的实现，继续向前。可以证明，一

个文档的所有格式扁平化为相同的串（见习题 E 答案）。浅嵌入还有另外一个性质，稍后在 pretty 的定义中讨论。为了认识这个性质，考虑返回一个格式的形状的函数 shape：

```
shape :: Layout -> [Int]
shape = map length . lines
```

引导库函数 lines 将一个串在换行符位置拆分，返回没有换行符的串的列表。所以，
188 一个格式的形状是组成格式的行的长度列表。Layouts 的关键性质是一个文档的格式形状列表是按照字典序递减排列的。例如，8.4 节中描述的一个文档有 13 种可能的格式，其形状如下：

```
[[94],[50,43],[50,28,19],[50,15,17,19],[10,39,43],
[10,39,28,19],[10,39,15,17,19],[10,28,15,43],
[10,28,15,28,19],[10,28,15,15,17,19],[10,13,19,15,43],
[10,13,19,15,28,19],[10,13,19,15,15,17,19]]
```

这个列表是按照字典序递减排列的。这个性质成立的原因是 layouts (group x)将扁平化的格式置于文档 x 的格式列表的前面，而且一个扁平化的格式只有一行。更多细节见习题 G。

8.4 例子

第一个例子是处理条件表达式的格式。就目前的目的，一个条件表达式可以表示成数据类型 CExpr 的元素，其中：

```
data CExpr = Expr String | If String CExpr CExpr
```

下面的函数 cexpr 说明本章开始描述的可接受格式：

```
cexpr :: CExpr -> Doc
cexpr (Expr p) = text p
cexpr (If p x y)
       = group (group (text "if " <> text p <>
                       line <> text "then " <>
                       nest 5 (cexpr x)) <>
                line <> text "else " <>
                nest 5 (cexpr y))
```

这个定义类似于前面的版本，只是子表达式的嵌套例外。

例如，下面显示一个特定表达式的 13 种可能格式的两种：

```
if wealthy
then if happy then lucky you else tough
189 else if in love then content else miserable
if wealthy
then if happy
     then lucky you
     else tough
else if in love
     then content
     else miserable
```

从最后一个表达式可以看出，为什么缩进量选择了 5 个空格。这个特定的条件表达式的 13 种可能格式的形状已在前一节展示了。

第二个例子有关如何显示一般的树，假定子树数目是任意的：

```
data GenTree a = Node a [GenTree a]
```

下面是一个树的例子，使用了两种不同的方式显示：

```
Node 1
  [Node 2
     [Node 7 [],
      Node 8 []],
   Node 3
     [Node 9
        [Node 10 [],
         Node 11 []]],
   Node 4 [],
   Node 5
     [Node 6 []]]

Node 1
  [Node 2 [Node 7 [], Node 8 []],
   Node 3 [Node 9 [Node 10 [], Node 11 []]],
   Node 4 [],
   Node 5 [Node 6 []]]
```

生成这些树（碰巧，以上格式也是13种可能格式的两种）的函数gtree定义如下：

```
gtree :: Show a => GenTree a -> Doc
gtree (Node x [])
    = text ("Node " ++ show x ++ " []")
gtree (Node x ts)
    =  text ("Node " ++ show x) <>
       group (nest 2 (line <> bracket ts))
```

190

定义的第一个子句表示，没有子树的树总是单行显示；第二个子句表示，至少有一棵子树的树或者单行显示，或者将每棵子树显示在新的一行，并缩进2个单位。函数bracket定义如下：

```
bracket :: Show a => [GenTree a] -> Doc
bracket ts = text "[" <> nest 1 (gtrees ts) <> text "]"

gtrees [t]    = gtree t
gtrees (t:ts) = gtree t <> text "," <> line <> gtrees ts
```

老实说，给出以上定义（对该定义函数layouts是不可或缺的）花了一点时间和实验，而且这个结果一定不是显示树的唯一方法。

最后，这里展示一种将一段文本（一个含空格和换行的串，不是文档text）显示为一个段落的方法：

```
para :: String -> Doc
para = cvt . map text . words

cvt [] = nil
cvt (x:xs)
    = x <> foldr (<>) nil [group (line <> x) | x <- xs]
```

首先，使用标准库函数words计算文本中的词，该函数以前曾多次见到；然后，使用text将每个词转换成文档；最后，除第一个词外，每个词显示在同一行或者新的一

行。如果文本中有 $n+1$ 个词，因此有 n 个分隔词的空格，以上代码描述了 2^n 种可能格式。在计算适合给定行宽的条件下，我们当然不想查看所有这些格式。

8.5 最佳格式

如上所述，最佳格式与最大允许行宽有关。这是一个简单的决定，但不是唯一的决
191 定。一般地，一个嵌套文档的美观格式表现为一条文本带蜿蜒跨越页面，而且有理由相信在选择最好格式时应该考虑带的宽度，即一行中除缩进外最多的字符数。总之，在一个无穷宽的页面上的最好格式表现为所有文本都在一行吗？不过，为简单起见，我们将忽略这个非常合理的改进，只把行宽作为决定因素。

还需要做出另外一个决定。假设根据某个标准，在所有满足行宽要求的格式中选择最好格式。如果至少存在一个这样的格式，那么选择没有问题，但是假如不存在怎么办？两个选择是，或者放弃格式化过程，返回一个错误信息，或者尽可能做出最好的选择，接受行宽超出限制。

从心理方面和实用方面讲，第二个选择似乎更好，所以我们将探讨这个选择的结果。下面从比较两个格式的第一行 ℓ_1 和 ℓ_2 开始。判断 ℓ_1 比 ℓ_2 好的条件：（ⅰ）两行都在行宽 w 范围内，而且 ℓ_1 比 ℓ_2 长；（ⅱ）ℓ_1 满足行宽要求，但是 ℓ_2 不满足；（ⅲ）两行均不在行宽 w 范围内，而且 ℓ_1 比 ℓ_2 短。这个决定是合理的，因为这个方法可以用贪心策略实现：在不超出行宽的条件下尽可能填满第一行；如果不可行，那么一旦超出行宽立即停止。

在两行长度相等的情况下，以上比较测试不能确定下一步如何进行。但是，因为所有格式扁平化为同一个串，如果第一行长度相等，则它们是同一行。因此，第一行已经确定，接着比较两个格式的第二行，如此反复下去。

关于形状按照字典序递减排列的性质也可用于简化比较测试，因为如果在格式列表中格式 lx 排在格式 ly 之前，那么 lx 的第一行长度大于等于 ly 第一行的长度。如果这两行长度相等，那么前一陈述句对第二行以及以后各行都是成立的。

对于以上给出的文档浅嵌入，下面是查找最优格式函数 `pretty` 的一个简单实现：

```
pretty :: Int -> Doc -> Layout
pretty w = fst . foldr1 choose . map augment
 where
 augment lx = (lx,shape lx)
 choose alx aly
  = if better (snd alx) (snd aly) then alx else aly
192 better [] ks          = True
 better js []          = False
 better (j:js) (k:ks) | j == k    = better js ks
                      | otherwise = (j <= w)
```

每种格式附上其形状信息用于指引格式的选择，这可用一个简单查找来完成。测试函数 `better` 实现以上描述的比较运算。最后，形状信息被丢弃。

函数 `pretty` 的这个定义是极度低效的，因为它要计算和查看每种格式。假如对于是否换行有 n 种可能的选择，那么需要检查 2^n 种格式，`pretty` 的运行将会极其慢。例如：

```
ghci> putStrLn $ pretty 30 $ para pg
This is a fairly short
paragraph with just twenty-two
words. The problem is that
pretty-printing it takes time,
in fact 31.32 seconds.
(31.32 secs, 17650013284 bytes)
```

天哪！更糟的是，格式打印更长的段落会使得 GHCi 崩溃，给出信息“内存耗尽”。任何指数时间和空间的算法都是不可接受的。

需要 `pretty` 算法最多查看 w 个字符便可确定第一行的选择。算法还应该是有效的，运行时间对于所处理文档大小是线性的。理想情况下运行时间不应该依赖于 w，但是如果更快的算法也意味着更复杂的程序，那么运算时间依赖于 w 也是可接受的。

8.6 项表示

将文档等同于它的可能格式列表的问题是结构信息的丢失。实际上将这个所有可能格式的列表隐藏得越深越好，而不应该把它提到最顶层。例如，考虑一个文档的以下两种表达式：

```
A<0>B<0>D + A<0>B<1>D + A<1>C<0>E + A<1>C<1>E
A(<0>B(<0>D + <1>D) + <1>C(<0>E + <1>E))
```

193

同前一样，`<0>`表示单个空格，`<1>`表示一个换行。其中 5 个字母表示 5 个非空文本。因为所有 4 种方法必须扁平化为同一个文档，故要求 `B<0>D = C<0>E`。在第一个表达式（基本上是用格式列表来表示文档）中有 4 个格式需要比较。在第二个式子中比较可以简化。例如，如果已知公共前缀 `A` 超过给定的行宽，那么前两个格式可以舍弃，无需进一步的比较。更进一步，如果从最内到最外选择，只需比较格式的第一行即可。例如，如果首先在 `C<0>E` 和 `C<1>E` 之间选择较好的，那么这个选择不会被以后的选择更改。

维护文档结构的方法是把文档表达成一棵树：

```
data Doc = Nil
         | Line
         | Text String
         | Nest Int Doc
         | Group Doc
         | Doc :<>: Doc
```

注意最后一行使用了中缀构造函数。Haskell 允许中缀运算符作为构造函数，但必须用冒号开始。结尾不必也用冒号，但是用冒号结尾显得更自然。这种树称为*抽象语法树*（abstract syntax tree），打印库的每个运算用一个构造函数表示。使用抽象语法树的实现便是所谓的*深嵌入*（deep embedding）。

打印库将为用户只提供数据类型 `Doc` 的名，不提供其他细节。为了说明这样做的原因，这里有必要就 Haskell 数据类型另外做一点说明。在 Haskell 中数据声明的作用是引入一个新的数据类型，方法是说明这个类型的值是如何构造的。每个值可以用仅由数据类型的构造函数构建的表达式命名，换句话说，每个值可用一个项（term）表示。而且，不同的项表示不同的值（只要不存在严格标志）。数据类型上的函数可以用构造函数上的模式

匹配定义。因此，无需说明数据类型上的运算是什么，只需给出定义即可。如果只描述类型的值，不描述类型的运算，这样的类型称为具体类型（concrete type）。

这种情况正好和抽象（abstract）数据类型相反。对于抽象数据类型，运算被命名，
194 但是对如何构造类型的值不做说明，至少没有公开给用户。例如，Float 是抽象数据类型，该类型提供了基本算术运算和比较运算的名，以及如何显示浮点数，但是并没有说明这些数是如何实际表示的。用户不可以用模式匹配的方法定义浮点数上的函数，只能用给定的运算定义其他函数。可以而且应该公开给用户的是这些运算的实际含义和代数性质。不过，Haskell 只提供非正式的注释，没有其他的说明方法。

如 Doc 所示，它是一个具体类型。但就我们对该类型的理解，不同的项并不表示不同的值。例如，每个构造函数的原意是代替相应的运算。因此，有

```
nil      = Nil
line     = Line
text s   = Text s
nest i x = Nest i x
group x  = Group x
x <> y   = x :<>: y
```

这些运算的代数性质也应该保持，如下面的等式应该成立：

```
(x :<>: y) :<>: z = x :<>: (y :<>: z)
Nest i (Nest j x) = Nest (i+j) x
```

但是这些等式当然不成立。解决问题的方法是利用模块结构把 Doc 的构造函数对用户隐藏起来，而且只认为这些定律是“观察上”正确的。例如，要求

```
layouts ((x :<>: y) :<>: z) = layouts (x :<>: (y :<>: z))
```

观察文档的唯一方法是通过 layouts。从用户的角度看，如果两个文档生成同一个格式，那么它们基本上是相同的文档。

现在返回到程序设计上来。下面是 layouts 的一种定义。该定义是先前看到的 layouts 满足的定律，只是现在表达成了适当的 Haskell 定义：

```
layouts :: Doc -> [Layout]
layouts (x :<>: y) = layouts x <++> layouts y
layouts Nil        = [""]
layouts Line       = ["\n"]
layouts (Text s)   = [s]
layouts (Nest i x) = map (nestl i) (layouts x)
layouts (Group x)  = layouts (flatten x) ++ layouts x
```
195

函数 flatten 的定义类似：

```
flatten :: Doc -> Doc
flatten (x :<>: y) = flatten x :<>: flatten y
flatten Nil        = Nil
flatten Line       = Text " "
flatten (Text s)   = Text s
flatten (Nest i x) = flatten x
flatten (Group x)  = flatten x
```

对于这些定义，24 条定律或者按照定义成立，或者在以上所述意义下观察正确。

函数 layouts 的定义非常简单，但是也包含不必要的低效。这里给出两个独立的理

由说明造成问题的原因。首先，考虑函数 egotist 的定义：

```
egotist :: Int -> Doc
egotist n | n==0      = nil
          | otherwise = egotist (n-1) <> text "me"
```

文档 egotist n 是非常枯燥的，而且它的唯一格式是由 n 个重复的 me 构成的串。顺便说明的是，可以使用 Nil、(:<>:)和 Text 来给出定义，但是，如前所讲，这些构造函数对用户不是公开的。如现在的情况，egotist 可能是打印库的用户定义的。总之，再返回讨论的问题，运算(<>)是左结合的，这使得计算它的格式需要$\Theta(n^2)$步。运算(++)被堆积到左面。这种情况完全类似于 concat 用 foldl 定义比用 foldr 定义效率低一个级别。

低效的第二个来源与嵌套有关。例如，考虑如下定义的函数 egoist：

```
egoist :: Int -> Doc
egoist n | n==0      = nil
         | otherwise = nest 1 (text "me" <> egoist (n-1))
```

定义中看不到换行，所以 egoist n 与 egotist n 表示同一个枯燥的文档。但是，尽管串联是右结合的，构造格式仍然需要二次方的运行时间。因为每个嵌套运算需要穿过整个文档才能完成。试试看。 196

解决第一个问题的方法是延迟串联，将一个串联的文档用其成分文档的列表表示。解决第二个问题的方法是延迟嵌套，嵌套文档用必要的缩进量和文档的二元组来表示。结合这两种解决方法，一个文档表示为缩进量和文档的二元组列表。特别地，考虑以下定义的函数 toDoc：

```
toDoc :: [(Int,Doc)] -> Doc
toDoc ids = foldr (:<>:) Nil [Nest i x | (i,x) <- ids]
```

现在可以给出函数 layr 的定义：

```
layr = layouts . toDoc
```

然后基于 layr 给出 layouts 的新定义。该定义细节留作练习，这里只给出结果：

```
layouts x = layr [(0,x)]
layr []                 = [""]
layr ((i,x :<>: y):ids) = layr ((i,x):(i,y):ids)
layr ((i,Nil):ids)      = layr ids
layr ((i,Line):ids)     = ['\n':replicate i ' ' ++ ls
                           | ls <- layr ids]
layr ((i,Text s):ids)   = [s ++ ls | ls <- layr ids]
layr ((i,Nest j x):ids) = layr ((i+j,x):ids)
layr ((i,Group x):ids)  = layr ((i,flatten x):ids) ++
                          layr ((i,x):ids)
```

这个定义对每个格式的运行时间是线性的。同样的模板用于选择单个最优格式函数pretty：

```
pretty w x = best w [(0,x)]
 where
 best r []                 = ""
 best r ((i,x :<>: y):ids) = best r ((i,x):(i,y):ids)
```

```
best r ((i,Nil):ids)      = best r ids
best r ((i,Line):ids)     = '\n':replicate i ' ' ++
                            best (w-i) ids
best r ((i,Text s):ids)   = s ++ best (r-length s) ids
best r ((i,Nest j x):ids) = best r ((i+j,x):ids)
best r ((i,Group x):ids)  = better r
                              (best r ((i,flatten x):ids))
                              (best r ((i,x):ids))
```

197

函数 best 的第一个参数是当前行剩余可用空间。该函数局部于函数 pretty 的定义以避免 best 附带最大行宽作为额外参数。

剩下的问题是计算 better r lx ly。可以利用 lx 的第一行长度一定不小于 ly 的第一行长度的事实。因此，只要比较 lx 第一行长度与 r 的大小即可。如果前者小于等于后者，则选择 lx，否则选择 ly。因此可以定义：

```
better r lx ly = if fits r lx then lx else ly
```

但是，我们不希望计算 lx 第一行整行长度，因为这样查看得太多。一种节俭的方法是

```
fits r _ | r<0 = False
fits r []      = True
fits r (c:cs)  = if c == '\n' then True
                 else fits (r-1) cs
```

基于同样的原因，better 的第二个和第三个参数使用惰性计算策略是很重要的，也就是说，两种格式的求值只需能够确定哪个更好即可，不需要计算更多。

再回过来看看有问题的段落：

```
ghci> putStrLn $ pretty 30 $ para pg
This is a fairly short
paragraph with just twenty-two
words. The problem is that
pretty-printing it takes time,
in fact 31.32 seconds.
(0.00 secs, 1602992 bytes)
```

现在看起来好多了。习题 L 讨论 pretty 的运行时间。

最后一个任务是将这个小函数库组织成一个模块。下面是主要的声明部分：

```
module Pretty
   (Doc, Layout,
    nil, line, text,
    nest, (<>), group,
    layouts, pretty, layout) where
```

198

模块名是 Pretty，而且包含这些声明和库函数定义的文件必须命名为 Pretty.lhs。

Pretty 模块输出 11 个对象。首先是抽象数据类型的名 Doc，该类型的构造函数没有输出。（顺便指出，如果确实想输出所有的构造函数，那么在输出列表中需要写 Doc(..)；如果只需要输出部分构造函数，如 Nil 和 Text，则需要写成 Doc(Nil,Text)。）然后是名 Layout，它是 String 的同义词。接下来是上面定义的 8 个常量和函数。最后的函数 layout 用于打印输出格式：

```
layout :: Layout -> IO ()
layout = putStrLn
```

现在格式打印库构建完成了。当然，对于一个真正实用的库，还需要另外提供一些组合函数。例如，可以提供：

```
(<+>),(<|>) :: Doc -> Doc -> Doc
x <+> y = x <> text " " <> y
x <|> y = x <> line <> y

spread,stack :: [Doc] -> Doc
spread = foldr (<+>) nil
stack  = foldr (<|>) nil
```

读者无疑可以想出许多其他的组合函数。

8.7 习题

习题 A 该库的一个挑剔用户只想要某个文档的 3 种格式：

```
A B C       A B       A
            C         B C
```

该用户能用库中输出的函数完成这个任务吗？

习题 B 一个文档的格式用一个列表给出。但是它们都各不相同吗？请给出证明，或者给出反例。另外，由定律是否可以明显地看出每个文档有一组非空的格式集。 199

习题 C 接下来的 4 个习题涉及 8.3 节的浅嵌入。用等式推理证明：

```
nest i . nest j  = nest (i + j)
```

这里可能需要有关 `nest1` 的辅助性质，但是不必证明。

习题 D 接着习题 C，用等式推理（在点层）证明：

```
nest i (group x) = group (nest i x)
```

这里同样需要一个辅助结果。

习题 E 接前习题 D，证明 `flatten . group = flatten`。这里也需要一个辅助结果。

习题 F 最后一个定律是 `flatten . nest i = flatten`。猜对了，这里也需要一个辅助结果。

习题 G 前面曾讲过，引导库函数 `lines` 将一个串按照换行符分成多个串。实际上，`lines` 把换行符看作终止字符，所以 `lines"hello"` 和 `lines"hello\n"` 返回相同的结果。一个更好的定义是将换行符看作分隔符（separator），这是有道理的，这样一来，行数总是比换行符多 1。请定义这样的函数 `lines`，以下将需要这个新的定义。

现在可以按下列步骤证明：将 `map shape` 应用于一个文档的格式返回一个按照字典序逆序排列的整数序列。首先，定义：

```
msl   = map shape . layouts
shape = map length . lines
```

200

其中 `lines` 是以上重写定义的函数。需要证明 `msl` 在任何文档上返回一个递减序列。为此，需要定义函数 `nesty` 和 `groupy` 使得

```
nesty i . msl = msl . nest i
groupy . msl  = msl . group
```

另外定义运算 <+> 使得

```
msl x <+> msl y = msl (x <> y)
```

（正是这个等式需要 lines 的新定义。）最后证明如果 xs 和 ys 是递减的，那么 nesty i xs 和 groupy xs 以及 xs < + > ys 都是递减的，由此完成证明。不过，本题只要求给出 nesty、groupy 和 < + > 的定义。

习题 H 定义一个函数 doc :: Doc -> Doc，该函数描述如何设置 Doc 元素的格式，其中 Doc 是 8.6 节的抽象语法树表示。

习题 I 考虑一个函数 prettybad，该函数从列表 layouts 中选择一个最好格式，其方法是选取所有行都适合给定行宽的第一个格式，如果不可行，则取最后一个。Prettybad是否总是计算出与 pretty 相同的结果？（提示：考虑文章的段落。）

习题 J 使用 Doc 构造函数的代数性质，计算 layouts 的高效定义。

习题 K 所设计的 pretty w 是最优的选择，也就是说，如有可能，则选择断行避免行超出给定行宽。pretty w 还是有界的，也就是说，无需查看输入的 w 字符即可选择下一行的断行位置。根据这些条件，判断下列 GHCi 命令的输出是什么？

```
layout $ pretty 5 $ para pg
layout $ pretty 10 $ cexpr ce
```

201 其中

```
pg = "Hello World!" ++ undefined
ce = If "happy" (Expr "great") undefined
```

习题 L 没有文档规模的定义就难以将 pretty w x 用 x 表达。下面是一个合理的规模定义：

```
size :: Doc -> Int
size Nil         = 1
size Line        = 1
size (Text s)    = 1
size (Nest i x)  = 1 + size x
size (x :<>: y)  = 1 + size x + size y
size (Group x)   = 1 + size x
```

按照这个定义，以下两个文档的规模都是 2。

```
nest 20 (line <> text "!")
nest 40 (line <> text "!")
```

但是，生成第二个格式的时间是生成第一个的 2 倍，所以 pretty 的代价不可能是文档规模的线性函数。

替代 pretty 生成最后格式即一个串的方法是，引入一个表示格式的数据类型：

```
data Layout = Empty
            | String String Layout
            | Break Int Layout
```

并定义 layout :: Layout -> String 如下：

```
layout Empty        = ""
layout (String s x) = s ++ layout x
layout (Break i x)  = '\n':replicate i ' ' ++ layout x
```

则有下列等式：

```
pretty w = layout . prettyl w
```

其中新的函数 prettyl 生成一个 Layout，而不是一个串。请定义 prettyl。

一个更合理的问题是，prettyl w x 是否关于 x 是线性的？ 202

8.8 答案

习题 A 答案 不能。不可能允许 A<0>B<1>C 和 A<1>B<0>C 而不允许 A<0>B<0>和 A<1>B<1>C。这 4 种格式由下式给出：

```
group (A <> line <> B) <> group (line <> C)
```

习题 B 答案 一个文档的格式不一定都不一样。例如：

```
layouts (group (text "hello")) = ["hello","hello"]
```

是的，显然每个文档都有一个非空格式集。观察有关 layouts 的定律，基本文档有一个非空格式列表，而且这个性质在其他运算下保持不变。

习题 C 答案 计算过程如下：

```
 nest i . nest j
=  {nest的定义}
 map (nestl i) . map (nestl j)
=  {map的函子律}
 map (nestl i . nestl j)
=  {辅助性质}
 map (nestl (i+j))
=  {nest的定义}
 nest (i+j)
```

辅助性质是 nestl i . nestl j = nestl (i+j)，可以根据下列等式做简单计算：

```
indent (i+j) = concat . map (indent i) . indent j
```

其证明省略。 203

习题 D 答案 推理如下：

```
 nest i (group x)
=  {group的定义}
 nest i (flatten x ++ x)
=  {因为nest i = map (nestl i)}
 nest i (flatten x) ++ nest i x
=  {辅助性质}
 flatten (nest i x) ++ nest i x
=  {group的定义}
 group (nest i x)
```

需要的辅助性质为

```
 nest i . flatten
=  {因为flatten x中不含换行}
 flatten
=  {因为flatten . nest i = flatten(习题F)}
 flatten . nest i
```

习题 E 答案 推理如下：

```
 flatten . group
=  {flatten和group的定义}
 one . flattenl . flattenl . head
=  {辅助性质}
 one . flattenl . head
=  {flatten的定义}
 flatten
```

辅助性质是 flattenl 满足幂等律：

```
flattenl . flattenl = flattenl
```

204 这是因为 flattenl 返回不含换行的格式。

另外，正是 flattenl 的幂等律保证一个文档扁平化为同一个串。唯一引入多个格式的函数是 group，其定义为

```
group x = flatten x ++ x
```

因此，必须证明对列表的第一个元素扁平化后得到的串等于对第二个元素扁平化后得到的串。故需要证明：

```
flattenl . head . flatten = flattenl . head
```

这个等式可以用 flatten 的定义和函数 flattenl 的幂等律推出。

习题 F 答案 推理如下：

```
 flatten . nest i
=  {定义}
 one . flattenl . head . map (nestl i)
=  {因为head . map f = f . head}
 one . flattenl . nestl i . head
=  {辅助性质}
 one . flattenl . head
=  {flatten的定义}
 flatten
```

需要的辅助性质是 flattenl . nestl i = flattenl。

习题 G 答案 可以定义：

```
lines xs = if null zs then [ys]
           else ys:lines (tail zs)
           where (ys,zs) = break (=='\n') xs
```

函数 groupy 定义如下：

```
groupy :: [[Int]] -> [[Int]]
groupy (xs:xss) = [sum xs + length xs - 1]:xs:xss
```

函数 nesty 定义如下：

205

```
nesty :: :: Int -> [[Int]] -> [[Int]]
nesty i = map (add i)
          where add i (x:xs) = x:[i+x | x <- xs]
```

函数(<+>)定义如下：

```
(<+>) :: [[Int]] -> [[Int]] -> [[Int]]
xss <+> yss = [glue xs ys | xs <- xss, ys <- yss]
 where glue xs ys = init xs ++ [last xs + head ys] ++
                    tail ys
```

习题 H 答案 一种定义如下，当然可以对其改进：

```
doc :: Doc -> Doc
doc Nil         = text "Nil"
doc Line        = text "Line"
doc (Text s)    = text ("Text " ++ show s)
doc (Nest i x) = text ("Nest " ++ show i) <>
                   group (nest 2 (line <> paren (doc x)))
doc (x :<>: y) = doc x <> text " :<>:" <>
                   group (line <> nest 3 (doc y))
doc (Group x)   = text "Group " <>
                   group (nest 2 (line <> paren (doc x)))

paren x = text "(" <> nest 1 x <> text ")"
```

习题 I 答案 不是。考虑这样的段落，其中最长的词比行宽多一个字符。prettybad 将把每个词放在单独一行，而 pretty 仍然可以将段落中适合行宽的一组词填入一行。例如：

```
ghci> putStrLn $ pretty 11 $ para pg4
A lost and
lonely
hippopotamus
went into a
bar.
```

206

习题 J 答案 首先证明 layouts x = layr [(0,x)]：

```
 layr [(0,x)]
=  {layr 的定义}
 layouts (toDoc [(0,x)])
=  {toDoc 的定义}
 layouts (Nest 0 x :<>: Nil)
=  {有关 Doc 的定律}
 layouts x
```

仍然需要给出 layr 的归纳定义。这里只给出两个子句：

```
 toDoc ((i,Nest j x):ids)
=  {toDoc 的定义}
 Nest i (Nest j x) :<>: toDoc ids
=  {定律}
 Nest (i+j) x :<>: toDoc ids
=  {toDoc 的定义}
 toDoc ((i+j x):ids)
```

所以有 layr ((i,Nest j x):ids) = layr ((i +j x):ids)。然后有

```
  toDoc ((i,x:<>:y):ids)
= {toDoc的定义}
  Nest i (x :<>: y) <> toDoc ids
= {定律}
  Nest i x :<>: Nest i y :<>: toDoc ids
= {toDoc的定义}
  toDoc ((i,x):(i,y):ids)
```

因此，layr ((i,x: < >:y):ids) = layr ((i,x):(i,y):ids)。

习题 K 答案

```
ghci> layout $ pretty 5 $ para pg
Hello
World1*** Exception: Prelude.undefined
ghci> layout $ pretty 10 $ cexpr ce
if happy
then great
else *** Exception: Prelude.undefined
```

207

习题 L 答案 函数 prettyl 的定义是

```
prettyl :: Int -> Doc -> Layout
prettyl w x = best w [(0,x)]
 where
 best r []                 = Empty
 best r ((i,Nil):ids)      = best r ids
 best r ((i,Line):ids)     = Break i (best (w-i) ids)
 best r ((i,Text s):ids)   = String s (best (r-length s) ids)
 best r ((i,Nest j x):ids) = best r ((i+j,x):ids)
 best r ((i,x :<>: y):ids) = best r ((i,x):(i,y):ids)
 best r ((i,Group x):ids)  = better r
                                (best r ((i,flatten x):ids))
                                (best r ((i,x):ids))
```

其中 better 定义改为

```
better r lx ly = if fits r (layout lx) then lx else ly
```

计算 better r 需要的化简步数正比于 r，因此最多是 w。

现在如果 best 是线性的，那么 prettyl 运行时间是线性的。best 的第二个参数是缩进和文档二元组列表，可以定义这个列表的规模为

```
isize ids = sum [size x | (i,x) <- ids]
```

对于 best 定义中间 5 个子句的每一个来说，其规模都减小了 1。例如：

```
isize ((i,x :<>: y):ids)
= size (x :<> y) + isize ids
= 1 + size x + size y + isize ids
= 1 + isize ((i,x):(i,y):ids)
```

由此推出，如果用 $T(s)$ 表示 best r 在规模为 s 的输入上的运行时间，那么由 best 的第一个子句得出 $T(0) = \Theta(1)$，对于中间 5 个子句有 $T(s+1) = \Theta(1) + T(s)$，而且对最后一个子句有

208

$$T(s+1) = \Theta(w) + \text{maximum}[T(k) + T(s-k) \mid k \leftarrow [1..s-1]]$$

现在可以推出 $T(s) = \Theta(ws)$。

总之，尽管算法 `pretty` 依赖于 w，但它是线性的。

8.9 注记

我们把美观打印库称为一个库，但是它的另一个名称是嵌入式领域专用语言（Embedded Domain Specific Language，EDSL）。这是一种嵌入 Haskell 用于美观打印文档的语言。许多人认为 Haskell 的不断成功来自于它很便于作为各种 EDSL 的宿主语言。

本章的详细内容主要基于 Philip Wadler 的工作，参见《The Fun of Programming in Cornerstones of Computing Series》（Palgrave MacMillan，2003）中的第 11 章“A prettier printer”。主要区别是，Wadler 在 `Doc` 的项表示中使用了显式的选择算子（不过对用户是隐藏的），而不是构造函数 `Group`。Jeremy Gibbons 建议后者更适合深嵌入。

更早的函数式美观打印库是 John Hughes 设计的“The design of a pretty-printer library”，该库基于一组不同的组合算子，参见 Johan Jeuring 和 Erik Meijer 等编辑的 Advanced Functional Programming，volume 925 of LNCS，Springer，1995。Simon Peyton Jones 对 Hughes 的库做了改造，并作为一个 Haskell 库安装为 `Text.PrettyPrint.HughesPJ`。

另一个命令式而不是函数式的美观打印库是 30 年前由 Derek Oppen 设计的“Pretty-printing”，参见 ACM Transactions on Programming Languages and Systems 2(4)，465-483，1980，而且该库是广泛用于多种语言美观打印的基础。最近，Olaf Chitil 提出了高效的函数式美观打印算法，参见 Pretty printing with lazy dequeues，ACM Transactions on Programming Languages and Systems 27(1)，163-184，2005，以及 Olaf Chitil、Doaitse 和 Swierstra 的 Linear，bounded，functional pretty-printing，Journal of Functional Programming 19(1)，1-16，2009。与本书介绍的算法相比，这些算法相当复杂。 209

第9章
Thinking Functionally with Haskell

无穷列表

我们在第4章已经遇到了无穷列表，而且在第6章给出了无穷列表上的归纳证明原理。但是，我们还没有真正认识无穷列表的作用。本章将详细解释什么是无穷列表，如何用循环（cyclic）结构表示它们。本章还将解释无穷列表上另一个有用的推理方法，并讨论一些有趣的例子，其中使用无穷列表和循环结构取得了很好的效果。

9.1 复习

回顾[m..]表示m之后所有整数的列表：

```
ghci> [1..]
[1,2,3,4,5,6,7,{Interrupted}
ghci> zip [1..] "hallo"
[(1,'h'),(2,'a'),(3,'l'),(4,'l'),(5,'o')]
```

输出[1..]需要无穷的时间，所以这里中断了第一个例子的计算。第二个例子表达了无穷列表的一个简单但是很典型的应用。

在Haskell中，算术表达式[m..]被翻译成enumFrom m，其中enumFrom是类族Enum的一个方法，定义为

```
enumFrom :: Integer -> [Integer]
enumFrom m = m:enumFrom (m+1)
```


因此，[m..]是递归定义函数的一个特例。因为(:)对第二个参数是非严格的，所以计算得以。

需要记住的是，无穷列表的计算与数学上无穷集合的计算不同。例如，在集合论中有

$$\{x \mid x \in \{1,2,3,\cdots\}, x^2 < 10\}$$

表示集合{1, 2, 3}，但是

```
ghci> [x | x <- [1..], x*x < 10]
[1,2,3
```

输出前三个值后，计算机不停地寻找3之后下一个平方小于10的数，因此陷入无限循环。上述表达式的值是非完整列表1:2:3:undefined。

构造无穷列表的无穷列表也是可行的。例如，下列表达式是无穷列表的无穷列表：

```
multiples = [map (n*) [1..] | n <- [2..]]
```

其中前三个是

```
[2,4,6,8,...]  [3,6,9,12,...]  [4,8,12,16,...]
```

假如我们想知道以上列表的列表是否可以归并为一个列表，即[2..]。归并两个无穷列表没有问题：

```
merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                    | x==y = x:merge xs ys
                    | x>y  = y:merge (x:xs) ys
```

以上 merge 删除了重复出现的元素：如果两个列表严格递增，那么结果也是严格递增的。注意 merge 没有一个子句说明空列表的情况。现在，如果定义

```
mergeAll = foldr1 merge
```

看似 mergeAll multiples 将返回无穷列表[2..]。但是，事情不是这样的。结果是计算机陷入尝试计算结果的第一个元素的无限循环中，即

```
minimum (map head multiples)
```

很简单，在一个无穷列表中求最小元是不可能的。所以，必须使用 map head multiples 是严格递增的事实，并定义 211

```
mergeAll = foldr1 xmerge
xmerge (x:xs) ys = x:merge xs ys
```

利用这个定义，mergeAll multiples 确实可以输出[2..]。

最后，回顾第 6 章描述的无穷列表上的归纳原理。只要 P 是一个链完全的性质，要证明 P(xs) 对于所有的无穷列表 xs 成立，只要证明：(i) P(undefined) 成立；(ii) 如果 P(xs) 成立，那么 P(x:xs) 对于所有 x 和 xs 成立。利用这个原则，我们在第 6 章证明了对于所有无穷列表 xs，xs ++ys = xs 成立。但是，并不是很清楚如何用归纳法证明如下等式：

```
map fact [0..] = scanl (*) 1 [1..]
```

明显地应该证明

```
map fact [0..n] = scanl (*) 1 [1..n]
```

对于所有 n 成立，但是，这样能够断定第一个等式成立吗？

9.2 循环列表

像函数一样，数据结构也可以递归定义。例如：

```
ones :: [Int]
ones = 1:ones
```

这是循环（cyclic）列表即递归定义的列表的一个例子。将这个定义与 ones = repeat 1 比较，其中：

```
repeat x = x:repeat x
```

ones 的这个定义构造了一个无穷列表，但不是一个循环列表。可以定义：

```
repeat x = xs  where xs = x:xs
```

现在函数 repeat 是用循环列表定义的。第二个定义（称为 repeat2）的计算比第一个（称为 repeat1）快，因为开销更少：

```
ghci> last $ take 10000000 $ repeat1 1
1
(2.95 secs, 800443676 bytes)
ghci> last $ take 10000000 $ repeat2 1
1
(0.11 secs, 280465164 bytes)
```

212

再看一个例子。考虑标准引导库函数 iterate 的下列三个定义：

```
iterate1 f x = x:iterate1 f (f x)
iterate2 f x = xs  where xs = x:map f xs
iterate3 f x = x:map f (iterate3 f x)
```

三个定义的类型均为(a -> a) -> a -> [a]，而且结果都是反复应用 f 于 x 的无穷列表。三个函数是相等的，但是早前的归纳原理似乎不能用来证明这个命题，因为找不到能够进行归纳的变量。后面会对此做进一步讨论。第一个定义是标准引导库的定义，但是这里并没有构造任何循环列表。第二个定义构造了循环列表，第三个定义只是在第二个定义中消去了 where 子句。假定 f x 可以用常数时间计算，第一个定义需要$\Theta(n)$步计算前n个元素，但是第三个定义需要$\Theta(n^2)$步：

```
  iterate3 (2*) 1
= 1:map (2*) (iterate3 (2*) 1)
= 1:2:map (2*) (map (2*) (iterate3 (2*) 1))
= 1:2:4:map (2*) (map (2*) (map (2*) (iterate3 (2*) 1)))
```

其中计算第n个元素需要(2*)的n次应用，所以生成前n个元素总共需要$\Theta(n^2)$步。

现在剩下第二个定义了。第二个定义需要线性时间还是二次方时间？表达式 iterate2 (2*) 1 的求值过程如下：

```
  xs       where xs = 1:map (2*) xs
= 1:ys     where ys = map (2*) (1:ys)
= 1:2:zs   where zs = map (2*) (2:zs)
= 1:2:4:ts where ts = map (2*) (4:ts)
```

生成结果的每个元素需要常数时间，所以 iterate2 (2*) 1 输出n个元素需要$\Theta(n)$时间。

现在构造一个循环列表，以生成所有素数的无穷列表。开始先定义：

```
primes     = [2..] \\ composites
composites = mergeAll multiples
multiples  = [map (n*) [n..] | n <- [2..]]
```

213

其中（\\）从一个严格递增的列表中减去另一个严格递增的列表：

```
(x:xs) \\ (y:ys) | x<y   = x:(xs \\ (y:ys))
                 | x==y  = xs \\ ys
                 | x>y   = (x:xs) \\ ys
```

这里 multiples 的构成是，从4开始的所有2的倍数构成的列表，从9开始的所有3的倍数构成的列表，从16开始的所有4的倍数构成的列表，等等。合并这些列表得到所有的合数构成的列表，然后关于[2..]取补得到所有素数。mergeAll 的定义在9.1节已给出。

到目前为止，一切顺利。但是，如果注意到太多的倍数被合并到合数中，那么算法通

过改进可以快许多倍。例如，已经构造了 2 的倍数，没必要再构造 4 的倍数，或者 6 的倍数等。我们真正想构造的是素数的倍数，由此导出“打一个递归结”的思想，定义：

```
primes = [2..] \\ composites
 where
 composites = mergeAll [map (p*) [p..] | p <- primes]
```

以上是 primes 的一个循环定义。定义看起来不错，但是可行吗？不幸的是，答案是否定的：primes 生成无定义列表。为了确定 primes 的第一个元素，计算过程需要得到 composites 的第一个元素，后者又需要 primes 的第一个元素。所以计算陷入无限循环中。要解决这个问题，必须显式给出第一个素数，从而从计算中抽出素数！因此需要如下改写定义：

```
primes = 2:([3..] \\ composites)
 where
 composites = mergeAll [map (p*) [p..] | p <- primes]
```

但是这个定义仍然不能输出素数！原因是细微的，而且不易发现。这和下面的定义有关：

```
mergeAll = foldr1 xmerge
```

肇事者是 foldr1 的定义。回顾它的 Haskell 定义：

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 xs)
```

214

定义的两个等式顺序很关键。特别是

```
foldr1 f (x:undefined) = undefined
```

因为列表参数首先与 x:[]匹配，由此导致结果为 undefined。这表示

```
mergeAll [map (p*) [p..] | p <- 2:undefined] = undefined
```

真正需要的是

```
mergeAll [map (p*) [p..] | p <- 2:undefined] = 4:undefined
```

为了得到这个效果，需要给 mergeAll 不同的定义：

```
mergeAll (xs:xss) = xmerge xs (mergeAll xss)
```

现在有

```
  mergeAll [map (p*) [p..] | p <- 2:undefined]
= xmerge (map (2*) [2..]) undefined
= xmerge (4:map (2*) [3..]) undefined
= 4:merge (map (2*) [3..]) undefined
= 4:undefined
```

这个 mergeAll 定义在有穷列表上的表现不同于前一个定义。为什么？

做了最后的修改后，可以说 primes 真正进入正轨了。但是如何证明这个结论呢？回答这个问题需要了解 Haskell 中关于递归定义的函数和值的语义，以及无穷列表如何定义为它们的非完整列表逼近的极限。

9.3 作为极限的无穷列表

在数学上某些值定义为更简单值的无限逼近序列的极限（limit）。例如，无理数

$$\pi = 3.14159265358979323846\cdots$$

可以定义为下列有理数无穷序列的极限：

215

$$3, 3.1, 3.14, 3.141, 3.1415, \cdots$$

第一个元素 3 是 π 的非常粗的逼近，下一个元素 3.1 是好一点的逼近，3.14 是更好的逼近，等等。

类似地，一个无穷列表也可以看成一个逼近序列的极限。例如，无穷列表 `[1..]` 是下列非完整列表的极限：

$$\bot, 1:\bot, 1:2:\bot, 1:2:3:\bot, \cdots$$

同样，序列是极限值越来越好的逼近。第一项 ⊥ 是无定义的元素，因此是非常粗的逼近：它没有极限的任何信息。下一个元素 1:⊥ 要好一点：它说明极限是一个列表，其第一个元素是 1，但是关于列表的尾部则没有任何信息。接下来的项 1:2:⊥ 更好一点，等等。每个更好的后续逼近都用另一个有定义的值代替 ⊥，因此给出极限的更多信息。

下面是另一个逼近序列，其极限是 `[1..]`：

$$\bot, 1:2:\bot, 1:2:3:4:\bot, 1:2:3:4:5:6:\bot, \cdots$$

该序列是前一个序列的子序列，但是它也收敛于同样的极限。

下面是一个不收敛于任何极限的逼近序列：

$$\bot, 1:\bot, 2:1:\bot, 3:2:1:\bot, \cdots$$

这个序列的问题在于它给出的信息是矛盾的：第二项说明极限的第一个元素是 1，但是，第三项说明极限的首元素是 2，第四项又说明极限的首元素是 3，等等。没有一个逼近说明它们的极限是什么，所以该序列不收敛。

不应该认为一个列表序列的极限必须是无穷的。例如，下列序列：

$$\bot, 1:\bot, 1:[\,], 1:[\,], \cdots$$

其中从第三项开始的每个元素都是 [1]，这是完全合理的序列，其极限为 [1]。类似地，

$$\bot, 1:\bot, 1:2:\bot, 1:2:\bot, \cdots$$

是以 1:2:⊥ 为极限的序列。有穷列表和非完整列表是只包含有穷个不同的元素序列的极限。

形式上表达非完整列表的无穷序列收敛于一个极限的方法是在每个类型的元素上引入

216

逼近序（approximation ordering）$\sqsubseteq$ 的概念。断言 $x \sqsubseteq y$ 表示 x 是 y 的一个逼近。序 $\sqsubseteq$ 是自反的（$x \sqsubseteq x$）、传递的（如果 $x \sqsubseteq y$ 并且 $y \sqsubseteq z$，那么 $x \sqsubseteq z$）和反对称的（如果 $x \sqsubseteq y$ 并且 $y \sqsubseteq x$，那么 $x = y$）。但是，该序并不要求每一对元素关于 $\sqsubseteq$ 可比较。因此，$\sqsubseteq$ 称为偏序（partial ordering）。注意，$\sqsubseteq$ 是一个数学符号（如 =），并不是一个返回布尔值的 Haskell 运算符。

数值、布尔、字符和任意其他枚举类型上的逼近序定义如下：

$$x \sqsubseteq y \equiv (x = \bot) \vee (x = y)$$

第一个子句表示 ⊥ 是所有元素的逼近。换言之，⊥ 是该序的底部（bottom）元素，简称底元。这也解释了为什么 ⊥ 读作“底部（bottom）”。值 ⊥ 是任何类型上逼近序 $\sqsubseteq$ 的底元。上面定义的序是平坦的（flat）。对于平坦序，我们或者知道一个值的一切信息，或者没有任何信息。

类型 (a, b) 上的序定义为 $\bot \sqsubseteq (x, y)$，而且

$$(x,y) \sqsubseteq (x',y') \equiv (x \sqsubseteq x') \wedge (y \sqsubseteq y')$$

右边$\sqsubseteq$的出现分别指类型 a 和类型 b 上的逼近序。类型 (a, b) 上的序$\sqsubseteq$不是平坦的，即使其分类型上的逼近序是平坦的。例如，在类型(Bool,Bool)上有下列不同元素间的链：

$$\bot \sqsubseteq (\bot, \bot) \sqsubseteq (\bot, \texttt{False}) \sqsubseteq (\texttt{True}, \texttt{False})$$

注意在 Haskell 中二元组 $(\bot, \bot)$ 不同于$\bot$：

```
ghci> let f (a,b) = 1
ghci> f (undefined,undefined)
1
ghci> f undefined
*** Exception: Prelude.undefined
```

类型[a]上的序$\sqsubseteq$定义为$\bot \sqsubseteq$ xs，(x:xs) $\not\sqsubseteq$[]，而且

$$[] \sqsubseteq xs \equiv xs = []$$

$$(x:xs) \sqsubseteq (y:ys) \equiv (x \sqsubseteq y) \wedge (xs \sqsubseteq ys)$$

这些等式应该看作一个数学断言的归纳定义，而不是 Haskell 定义。第二个条件表示[]只是自己的逼近，第三个条件表示(x:xs)是(y:ys)的逼近当且仅当 x 是 y 的逼近，而且 xs 是 ys 的逼近。定义右边$\sqsubseteq$的第一次出现表示类型 a 上的逼近序。 217

以下是两个逼近的例子：

$$[1, \bot, 3] \sqsubseteq [1,2,3] \quad \text{而且} \quad 1:2:\bot \sqsubseteq [1,2,3]$$

但是，1:2:⊥和 [1, ⊥, 3] 关于$\sqsubseteq$不可比较。

每个类型 T 上的逼近序除以上描述的性质外，还有另外一个性质：每个逼近链(chain) $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$必须在 T 中具有一个极限。用$\lim_{n\to\infty} x_n$表示的该极限由下列两个条件定义：

1. $x_n \sqsubseteq \lim_{n\to\infty} x_n$ 对于所有的 n 成立。这个条件表示极限是逼近序列的上界（upper bound）。
2. 如果对于所有的 n 有 $x_n \sqsubseteq y$，那么$\lim_{n\to\infty} x_n \sqsubseteq y$。这个条件表示极限是最小上界。

逼近链的极限定义适用于任何类型。具有这种性质的偏序称为完全的（complete），而且每个 Haskell 类型是一个完全偏序（Complete Partial Ordering，CPO）。特别是，第 6 章引入的 P 的链完全性质断言现在可以描述为

$$(\forall n : P(x_n)) \Rightarrow P(\lim_{n\to\infty} x_n)$$

换句话说，如果 P 对于极限的每个逼近成立，那么 P 对于极限成立。

Haskell 有一个关于列表的有用函数 approx，该函数生成一个给定列表的逼近。其定义如下：

```
approx :: Integer -> [a] -> [a]
approx n []     | n>0 = []
approx n (x:xs) | n>0 = x:approx (n-1) xs
```

函数 approx 的这个定义非常类似于 take，区别在于分情况定义，对于任意 xs，有 approx 0 xs = undefined。例如：

```
approx 0 [1] = undefined
approx 1 [1] = 1:undefined
approx 2 [1] = 1:[]
```

approx 最关键的性质是

218

$$\lim_{n\to\infty} \texttt{approx n xs} = \texttt{xs}$$

对于任意列表 xs，包括有穷、非完整和无穷列表都成立。证明可以在 xs 上归纳进行，留作练习。

由此得出，如果 approx n xs = approx n ys 对于所有的自然数 n 都成立，那么 xs = ys。因此可以通过证明

```
approx n (iterate f x) = approx n (x:map f (iterate f x))
```

对于所有的自然数 n 成立来证明

```
iterate f x = x:map f (iterate f x)
```

当然，前一个等式的证明可以对自然数进行归纳。证明留给读者作为练习。

再来看一个例子，考虑9.2节定义的 primes。假如定义：

```
prs n = approx n primes
```

要想证明 prs n = p_1: p_2: $\cdots p_n$: ⊥，其中 p_j 是第 j 个素数，则需证明：

```
prs n = approx n (2:([3..] \\ crs n))
crs n = mergeAll [map (p*) [p..] | p <- prs n]
```

利用这些条件，可以证明 crs n = c_1: c_2: $\cdots c_m$: ⊥，其中 c_j 是第 j 个合数（如 $c_1 = 4$）而且 $m = p_n^2$。然后可以利用不等式 $p_{n+1} < p_n^2$ 完成证明，只是这个不等式是数论中的一个非平凡结论。详情见习题。

可计算函数与递归定义

我们可以描述许多数学函数，但是只有一部分是可计算的。可计算函数有两个性质是其他函数不具有的。第一，一个可计算函数关于逼近序是单调的（monotonic）。用符号来表示，对于所有的 x 和 y，有

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

粗略地讲，单调性表示对于参数提供的信息越多，得到的结果的信息也越多。第二，一个可计算函数是连续的（continuous），也就是说

219

$$f(\lim_{n\to\infty} x_n) = \lim_{n\to\infty} f(x_n)$$

对于所有的逼近链 $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ 成立。连续性大致表示，如果参数有极限，那么结果也有极限。

连续性看似与链完全性相似，但是它们在两方面不同。一方面，P 的链完全性并不意味着如果 P 对于所有逼近不成立，则 P 对其极限也不成立。换言之，它并不能推出 $\neg P$ 是链完全的。另一方面，P 是一个数学命题，不是返回布尔值的 Haskell 函数。

这里不加证明但给出一个结果：每个单调且连续的函数 f 具有一个最小不动点（least fixed point）。一个函数 f 的不动点是使得 $f(x) = x$ 的值 x。称 x 是最小不动点，如果对于所有的其他不动点 y 有 $x \sqsubseteq y$。一个单调连续函数 f 的不动点由极限 $\lim_{n\to\infty} x_n$ 给出，其中 $x_0 = \bot$，$x_{n+1} = f(x_n)$。在函数式程序设计中，递归定义被解释为最小不动点。

下面是三个例子。第一个例子考虑定义 ones = 1:ones，该定义表示 ones 是函数

(1:)的一个不动点。Haskell 将其解释为最小不动点，所以 $\texttt{ones} = \lim_{n\to\infty} \texttt{ones}_n$，其中 $\texttt{ones}_0 = \bot$，$\texttt{ones}_{n+1} = 1 : \texttt{ones}_n$。容易看出，$\texttt{ones}_n$ 是 n 个 1 构成的非完整列表，所以其极限确实是无穷个 1 的列表。

第二个例子考虑阶乘函数：

```
fact n = if n==0 then 1 else n*fact (n-1)
```

可以将该函数改写为下面的等价形式

```
fact = (\f n -> if n==0 then 1 else n*f(n-1)) fact
```

同样，这个函数说明 fact 是一个函数的不动点。对这个例子，有

```
fact0 n = ⊥
fact1 n = if n==0 then 1 else ⊥
fact2 n = if n<=1 then 1 else ⊥
```

等等。如果 n 小于 k，那么 $\texttt{fact}_k n$ 的值是 n 的阶乘，否则是 $\bot$。

第三个例子再次考虑列表 primes。对这个例子，有

```
primes0   = ⊥
primesn+1 = 2:([3..] \\
            mergeAll [map (p*) [p..] | p <- primesn])
```

220

这里并不是指 $\texttt{primes}_n = \texttt{approx n primes}$。事实上，有

$\texttt{primes}_1 = 2 : \bot$

$\texttt{primes}_2 = 2 : 3 : \bot$

$\texttt{primes}_3 = 2 : 3 : 5 : 7 : \bot$

$\texttt{primes}_4 = 2 : 3 : 5 : 7 : \cdots : 47 : \bot$

非完整列表 $\texttt{primes}_2$ 生成小于 4 的所有素数，$\texttt{primes}_3$ 生成小于 9 的所有素数，$\texttt{primes}_4$ 生成小于 49 的所有素数，等等。

9.4 石头-剪刀-布

下一个无穷列表的例子可谓寓教于乐。这个例子不仅介绍使用潜在无穷列表来表示两个进程间的交互序列的思想，而且提供了另一个说明做形式化分析必要性的具体例子。

石头-剪刀-布游戏是妇孺皆知的游戏，只是不同的地方它的名称可能不同。游戏在两个玩家之间面对面进行，每个玩家在自己背后做出形如石头（紧握的拳头）或者布（伸展的手掌）或者剪刀（两个伸开的手指）的手势，在约定的时刻同时亮出藏在背后的手势。输赢规则为“布包石头，石头钝剪刀，剪刀剪布”。因此，如果玩家 1 伸出石头，玩家 2 伸出剪刀，那么玩家 1 赢，因为石头可以钝剪刀。如果两个玩家伸出同一种手势，那么结果是平局，没有赢家。游戏以这种方式连续进行事先约定数目的回合。

本节的目标是设计一个玩这种游戏并记录分数的程序。首先从引入类型开始：

```
data Move  = Paper | Rock | Scissors
type Round = (Move,Move)
```

给一个回合计分，定义：

```
score :: Round -> (Int,Int)
score (x,y) | x `beats` y  = (1,0)
            | y `beats` x  = (0,1)
            | otherwise    = (0,0)
```

221 其中：

```
Paper `beats` Rock     = True
Rock `beats` Scissors  = True
Scissors `beats` Paper = True
_ `beats` _            = False
```

游戏的每个玩家将被表示成某种策略。例如，一种简单策略是，在第一个回合后每次选择对手上一个回合的出手。这个策略称为 copy。另一策略基于一定的分析和计算做出回应，称为 smart，其方法是通过分析对手已经做出的每种出手的次数，基于一定的概率计算适当的下一次出手。

稍后将考虑特定策略的细节，以及这些策略如何表示。目前先假定类型 Strategy 用某种方式给出。引入函数：

```
rounds :: (Strategy,Strategy) -> [Round]
```

对于一对策略，该函数返回当每个玩家根据其策略出手时的无穷回合列表。下列函数决定给定轮次猜拳后的成绩：

```
match :: Int -> (Strategy,Strategy) -> (Int,Int)
match n = total . map score . take n . rounds
  where total rs = (sum (map fst rs),sum (map snd rs))
```

该游戏中具有教育意义的是如何表示策略。考虑两种表示方式，分别称为 Strategy1 和 Strategy2。最明显的想法是定义：

```
type Strategy1 = [Move] -> Move
```

这里策略表示成一个函数，该函数的输入是对手到目前为止的出手（有穷）列表，返回值是下一轮的出手。为了处理列表方便，假定出手的列表用逆序列出，即最后的出手是列表的第一个元素。

例如，策略 copy1 是如下实现的：

```
copy1 :: Strategy1
copy1 ms = if null ms then Rock else head ms
```

222 第一个出手是随意选择的 Rock，第二个策略 smart1 如下定义：

```
smart1 :: Strategy1
smart1 ms = if null ms then Rock
            else pick (foldr count (0,0,0) ms)

count :: Move -> (Int,Int,Int) -> (Int,Int,Int)
count Paper (p,r,s)    = (p+1,r,s)
count Rock  (p,r,s)    = (p,r+1,s)
count Scissors (p,r,s) = (p,r,s+1)

pick :: (Int,Int,Int) -> Move
pick (p,r,s)
     | m < p     = Scissors
     | m < p+r   = Paper
     | otherwise = Rock
     where m = rand (p+r+s)
```

这个策略首先计算每种手势出现的次数，然后根据这个结果选择一种手势。Rand 应用于 n 返回一个整数 m，而且 $0 \leqslant m < n$。（注意 rand 永远不会应用于同一个整数。）所以，最后做出的选择依赖于 m 落入下面三个区间中哪一个：

$$0 \leqslant m < p \quad \text{或者} \quad p \leqslant m < p+r \quad \text{或者} \quad p+r \leqslant m < p+r+s$$

例如，如果 p 很大，那么 Scissors 被选中的概率很大（因为剪刀剪布）；如果 r 很大，那么 Paper 被选中的概率大（因为布包石头）；等等。

为定义 rand，利用库 System.Random 的两个函数：

```
rand :: Int -> Int
rand n = fst $ randomR (0,n-1) (mkStdGen n)
```

函数 mkStdGen 输入一个整数，返回一个随机数生成器，不同的整数可能得到不同的生成器。mkStdGen 的参数选择是任意的，这里简单取 n。函数 randomR 取一个区间 (a, b) 与一个随机数生成器，返回区间 $a \leqslant r \leqslant b$ 上的一个伪随机数 r 和一个新的随机数生成器。

现在可以定义 rounds1：

```
rounds1 :: (Strategy1,Strategy1) -> [Round]
rounds1 (p1,p2)
        = map head $ tail $ iterate (extend (p1,p2)) []
extend (p1,p2) rs = (p1 (map snd rs),p2 (map fst rs)):rs
```

223

函数 extend 在现有回合列表前添加一对新的出手，rounds1 在初始的空回合列表上不断添加新的回合，从而生成无穷的回合列表。在列表前添加元素比在尾部添加元素更高效，这是回合列表用逆序表示的原因。

不过 rounds1 的效率不高。假设一个策略需要正比于其输入长度的时间计算下一个出手。由此得出 extend 需要 $\Theta(n)$ 步用一个新回合更新 n 个回合的游戏。因此，计算 N 个回合的游戏需要 $\Theta(N^2)$ 步。

为了进行比较，考虑另一个策略的合理表示。这次定义：

```
type Strategy2 = [Move] -> [Move]
```

在新的表示中，策略是一个函数，函数的输入是对手的出手潜在无穷列表，输出是回应对手的潜在出手无穷列表。例如，copy 策略现在可以这样实现：

```
copy2 :: Strategy2
copy2 ms = Rock:ms
```

该策略首次返回 Rock，以后每次都返回对手前一回合的出手。smart 策略如下实现：

```
smart2 :: Strategy2
smart2 ms = Rock:map pick (stats ms)
 where stats = tail . scanl (flip count) (0,0,0)
```

其中函数 stats 计算 3 种手势出现的次数。像 copy2 一样，这个策略也是高效的，它可以在常数时间返回每个后续的出手。

使用新的策略模型，函数 rounds 可以重定义如下：

```
rounds2 :: (Strategy2,Strategy2) -> [Round]
rounds2 (p1,p2) = zip xs ys
                  where xs = p1 ys
                        ys = p2 xs
```

这里 xs 是第一个玩家根据列表 ys 计算出的应对出手列表，而 ys 是第二个玩家根据列表 xs 计算出的应对出手列表。因此，rounds2 是用两个循环列表定义的，并且
224 必须说明该函数确实生成良好定义的回合的无穷列表。这点稍后再做说明。如果两个玩家真正使用合法的策略，每个玩家用常数时间计算出下一次出手，那么 rounds2 用$\Theta(n)$步计算出游戏的前 n 个回合出手。因此，使用策略的第二个模型得到更高效的程序。

不幸的是，策略的第二种表示存在严重缺陷：它没有提供防欺骗的方法！考虑下面的策略：

```
cheat ms = map trump ms

trump Paper    = Scissors
trump Rock     = Paper
trump Scissors = Rock
```

cheat 的第一次回应是确保打赢对手第一次出手的手势，随后的出手也类似。为了看清楚不能阻止 cheat 毁掉游戏，考虑它与策略 copy2 的比赛，并令 xs = cheat ys，ys = copy2 xs。列表 xs 和 ys 分别是两个链 $\{xs_n \mid 0 \leqslant n\}$ 和 $\{ys_n \mid 0 \leqslant n\}$ 的极限，其中 $xs_0 = \bot$，xs_{n+1} = cheat ys_n，$ys_0 = \bot$，ys_{n+1} = copy2 xs_n。现在有

```
xs1 = cheat ⊥           = ⊥
ys1 = copy2 ⊥           = Rock: ⊥
xs2 = cheat (Rock: ⊥)   = Paper:⊥
ys2 = copy2 ⊥           = Rock:⊥
xs3 = cheat (Rock: ⊥)   = Paper:⊥
ys3 = copy2 (Paper: ⊥)  = Rock:Paper:⊥
```

按照这种方式重复下去，可以看出这些序列的极限确实是良好定义的出手列表的极限。而且，cheat 总是赢。另一种欺骗方法如下：

```
devious :: Int -> Strategy2
devious n ms = take n (copy2 ms) ++ cheat (drop n ms)
```

这个策略前 n 个出手像拷贝，然后开始欺骗。

是否可以找到一种防止欺骗的方法？要回答这个问题，需要仔细看看什么是诚实策略。非正式地讲，如果一个策略在计算其第一出手时没有使用对手第一次出手的任何信息，计算第二次出手时也没有使用对手第二次出手的任何信息，等等，那么这个策略是诚实的。而且，假定对手的出手是良好定义的，那么每次出手都是良好定义的。更确切地
225 说，假设用 $wdf(n, ms)$ 表示出手列表 ms（可能非完整）的前 n 个元素是良好定义的。如果对于所有 n 和 ms 都有下列命题成立：

$$wdf(n, ms) \Rightarrow wdf(n+1, f(ms))$$

则称策略 f 是诚实的。容易证明，copy2 是诚实的。另一方面，因为 $wdf(0, \bot)$ 真，但是 $wdf(1,$ cheat $\bot)$ 不真，所以 cheat 是不诚实的。再看策略 dozy：

```
dozy ms = repeat undefined
```

尽管该策略不存在欺骗，但是根据定义它是不诚实的。

找到不合法或者无意义行为的根源后，能否确保只有诚实的策略才可以玩游戏？答

案是有条件的“是”：尽管一个机械的计算器不可能识别欺骗（同样不能识别⊥或者一个策略是否返回良好定义的出手），但是可以定义一个函数 police，使得如果 p 是一个诚实的玩家，ms 是良好定义的出手无穷序列，那么 police p ms = p ms。另一方面，如果 p 在某时刻是不诚实的，那么游戏在该时刻以⊥终止。从操作语义上讲，police 强迫 p 在它得到其输入前返回其输出的第一个（良好定义的）元素，对于其他元素也类似地进行。其定义如下：

```
police p ms = ms'  where ms' = p (synch ms ms')
synch (x:xs) (y:ys) = (y `seq` x):synch xs ys
```

回顾第 7 章 x `seq` y 的求值，在返回 y 的值之前需要先对 x 求值。证明这个定义满足其规格说明需要很大的投入，故不在此详述。从上面的分析可以得出，要避免欺骗必须重新改写 rounds2 的定义如下：

```
rounds2 (p1,p2) = zip xs ys
                  where xs = police p1 ys
                        ys = police p2 xs
```

9.5 基于流的交互

在石头-剪刀-布游戏中，使用了一个输入为出手的无穷列表并返回类似列表的函数表示交互。同样的思想可用于建立输入-输出交互的简单模型，称为*基于流*（stream-based）的交互，因为无穷列表也称为流。Haskell 提供了一个与外界交互的函数：

```
interact :: ([Char] -> [Char]) -> IO ()
```

226

interact 的参数是一个函数，其输入是来自标准输入的潜在无穷的字符列表，并返回打印在标准输出的一个潜在无穷的字符列表。例如：

```
ghci> import Data.Char
ghci> interact (map toUpper)
hello world!
HELLO WORLD!
Goodbye, cruel world!
GOODBYE, CRUEL WORLD!
{Interrupted}
```

首先输入库 Data.Char 以便可以使用 toUpper，然后创建一个将每个字符转换为大写的交互。每次输入一行（重复出现在显示器上），交互程序输出全部大写的输入行。该进程连续运行直至用户中断它。

也可以设计一个终止的交互。例如：

```
interact (map toUpper . takeWhile (/= '.'))
```

该程序实现类似于以上的交互，而且当输入行键入句点后立即终止：

```
ghci> interact (map toUpper . takeWhile (/= '.'))
Goodbye. Forever
GOODBYE
```

下面是一个能够独立运行的程序，它读取一个 Haskell 文学型程序作为输入，然后返回将首字符不是 > 的非空行删除，其余行的首字符 > 被删除的文件。所以结果是一个合法

的.hs 文件（非文学型的 Haskell 脚本）：

```
main    = interact replace
replace = unlines . map cleanup . filter code . lines
code xs = null xs || head xs == '>'
cleanup xs = if null xs then [] else tail xs
```

该程序是与标识符 main 相关联的计算，而且任何程序如果要进行编译，必须给出 main 的定义。函数 lines 将文本分解成行，unlines 在这些行间添加换行符后重新将其合成
227 文本。如果将程序存储为 lhs2hs.lhs，那么可以将其编译，然后运行：

```
$ ghc lhs2hs.lhs
$ lhs2hs <myscript.lhs >myscript.hs
```

其中第二行的输入来自 myscript.lhs，输出重定向到 myscript.hs。

在 Haskell 早期版本中，基于流的交互是与外界交互的主要方法。但是，以上解释的模型对于大多数实际程序设计来说过于简单。在更实际的应用中，需要比从键盘读字符和在屏幕上写字符更多的交互。例如，需要打开文件并读文件、写文件，或者删除文件，总之需要与一个函数语言范围之外的所有设备和机制交互。交互是实时发生的，程序员必须正确地处理交互事件发生的次序。在基于流的模型中，事件的次序用列表中元素的次序表示。换言之，次序在数据中表达，基本没有反映在程序编写的方式上。第 10 章将考虑另一种交互的方法——一种能真正用于控制事件序列顺序程序的通用方法。使用这种方法时，事件的次序显式地反映在程序的编写方式中。

9.6 双向链表

下面用循环列表的另一个应用结束本章。设想阅读由页的非空列表组成的一本书。在书中浏览时，需要有移到下一页或者翻到前一页的方法。其他的浏览工具也是有用的，不过这里只考虑这两种方法。下面是与一本无聊的只由 3 页构成的书的交互过程：

```
ghci> start book
"Page 1"
ghci> next it
"Page 2"
ghci> prev it
"Page 1"
ghci> next it
"Page 2"
ghci> next it
228 "Page 3"
```

在 GHCi 中变量 it 被绑定到刚刚在提示符后键入的表达式上。展开一本书时，打印出来的是第一页。翻到下一页，然后返回前一页。有趣的问题是，当我们已经在最后一页时，翻到下一页应该如何处理？浏览程序应该报错，仍然停留在最后一页，还是转到第一页？假如选择后者，最后一页的下一页是第一页，第一页的前一页是最后一页。换言之，书是*循环双链表*（cyclic doubly-linked list）的一个例子。

下面是有关数据类型的声明：

```
data DList a = Cons a (DList a) (DList a)

elem :: DList a -> a
```

```
elem (Cons a p n) = a

prev,next :: DList a -> DList a
prev (Cons a p n) = p
next (Cons a p n) = n
```

打印双链表通过显示当前元素来完成：

```
instance Show a => Show (DList a)
  where show d = show (elem d)
```

这样一来，上面提到的书就是一个3页列表[p1,p2,p3]，其中：

```
p1 = Cons "Page 1" p3 p2
p2 = Cons "Page 2" p1 p3
p3 = Cons "Page 3" p2 p1
```

这个例子表明将一个（非空）列表as转换为双链表的函数mkCDList :: [a] -> DList a可以说明为一个双链表的有穷列表xs的第一个元素，并且满足下列3个性质：

```
map elem xs = as
map prev xs = rotr xs
map next xs = rotl xs
```

其中rotr和rotl（分别是右转rotate right和左转rotate left的简写）定义如下：

```
rotr xs = [last xs] ++ init xs
rotl xs = tail xs ++ [head xs]
```

229

注意现在对于任意双链表的列表xs，有

```
xs = zipWith3 Cons
     (map elem xs) (map prev xs) (map next xs)
```

其中zipWith3的功能如zipWith，只是现在它需要取3个列表，而不是2个列表。标准引导库的定义为

```
zipWith3 f (x:xs) (y:ys) (z:zs)
    = f x y z : zipWith3 f xs ys zs
zipWith3 _ _ _ _ = []
```

稍后将看到另一个定义。可以用归纳法证明前面的断言，对于无定义值和空列表显然成立。对于归纳情况，推理如下：

```
  x:xs
=  {因为xs是双链表}
  Cons (elem x) (prev x) (next x):xs
=  {归纳假设}
  Cons (elem x) (prev x) (next x):
  (zipWith3 Cons
  (map elem xs) (map prev xs) (map next xs))
=  {zipWith3和map的定义}
  zipWith3 Cons
  (map elem (x:xs)) (map prev (x:xs)) (map next (x:xs)
```

结合这个结果与双链表的说明，得到

```
mkCDList as = head xs
  where xs = zipWith3 Cons as (rotr xs) (rotl xs)
```

这个定义涉及一个循环列表 xs。这样可行吗？答案是“不可行”。原因是如上定义的 zipWith3 太勤奋。需要使它懒一点，当另外两个列表不是真正需要时不对它们求值：

```
zipWith3 f (x:xs) ys zs
    = f x (head ys) (head zs):
      zipWith3 f xs (tail ys) (tail zs)
zipWith3 _ _ _ _ = []
```

230

定义这个函数的一个等价方法是利用 Haskell 的*无争辩模式*（irrefutable pattern）：

```
zipWith3 f (x:xs) ~(y:ys) ~(z:zs)
    = f x y z : zipWith3 f xs ys zs
zipWith3 _ _ _ _ = []
```

无争辩模式用一个波浪号表示，~(x:xs)的匹配是惰性的，即匹配是在 x 或者 xs 被需要时才进行。

为了确保以上使用 zipWith3 修改版的 mkCDList 定义确实取得进展，令 $xs_0 = \bot$，并且令

$$xs_{n+1} = \texttt{zipWith3 Cons "A" (rotr } xs_n\texttt{)(rotl } xs_n\texttt{)}$$

然后 xs_1 如下：

```
zipWith3 Cons "A" ⊥ ⊥
= [Cons 'A' ⊥ ⊥]
```

xs_2 如下：

```
zipWith3 Cons "A"
[Cons 'A' ⊥ ⊥] [Cons 'A' ⊥ ⊥]
= [Cons 'A' (Cons 'A' ⊥ ⊥) (Cons 'A' ⊥ ⊥)]
```

等等。

9.7 习题

习题 A 给定 3 个严格递增的列表 xs、ys 和 zs，则有

```
merge (merge xs ys) zs} = merge xs (merge ys zs)
```

因此，merge 满足结合律。另外假定 xs、ys 和 zs 的第一个元素是严格递增的，那么还有

```
xmerge (xmerge xs ys) zs = xmerge xs (xmerge ys zs)
```

由此是否可以推出表达式 foldr1 xmerge multiples 中的 foldr1 可以用 foldl1 代替？

231

习题 B 标准引导库函数 cycle :: [a] -> [a]取一个列表 xs，返回 xs 元素的无穷次重复列表。如果 xs 是空列表，那么 cycle []返回错误信息。例如：

```
cycle "hallo" = "hallohallohallo..."
```

请使用一个循环列表定义 cycle，确保该定义在空列表、有穷列表和无穷列表上都可行。

习题 C 斐波那契函数定义如下：

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

请用一行定义写出能够生成斐波那契数列的无穷列表 fibs。

习题 D 一个源于数学家 W. R. Hamming 的有名问题是设计一个程序，该程序能够生成一个数的无穷列表，并且满足性质：（ⅰ）列表严格递增；（ⅱ）列表从 1 开始；（ⅲ）如果列表包含数 x，那么列表也包含 $2x$、$3x$ 和 $5x$；（ⅳ）列表不含其他数。因此，列表以下列这些数开始：

$$1,2,3,4,5,6,8,9,10,12,15,16,\cdots$$

请给出生成这个列表 hamming 的定义。

习题 E 证明 approx n xs ⊑ xs 对于所有的 n 都成立。然后证明如果 approx n xs ⊑ ys 对所有的 n 成立，那么 xs ⊑ ys。因此可以得出如下结论：

$$\lim_{n\to\infty} \text{approx n xs} = \text{xs}$$

习题 F 给出断言的反例：如果对于所有的 n 有 xs!! n = ys!! n，那么 xs = ys。 232

习题 G 证明 iterate f x = x: map f (iterate f x)。

习题 H 在 primes 的循环列表定义中，是否可以用定义

```
mergeAll = foldr xmerge []
```

来替代书中的定义？

习题 I 回顾定义：

```
prs n = approx n (2:([3..] \\ crs n))
crs n = mergeAll [map (p*) [p..] | p <- prs n]
```

假定 prs n $= p_1 : p_2 : \cdots : p_n : \perp$，其中 p_j 是第 j 个素数，请说明如何证明 crs n $= c_1 : c_2 : \cdots : c_m : \perp$，其中 c_j 是第 j 个合数（如 $c_1 = 4$），而且 $m = p_n^2$。给出证明梗概即可。由此证明 primes 确实可以生成素数的无穷列表。

9.3 节曾经讲过，primes 的第 n 个逼近 primes_n 不等于 approx n primes。事实上，有

$$\text{primes}_4 = 2:3:5:7:\cdots:47:\perp$$

请问 primes_5 生成什么？

习题 J 另一种生成素数的方法是所谓的 Sundaram 筛法，是数学家 S. P. Sundaram 在 1934 年发现的：

```
primes   = 2:[2*n+1 | n <- [1..] \\ sundaram]
sundaram = mergeAll [[i+j+2*i*j | j <- [i..]] | i <- [1..]]
```

要证明 primes 定义中的列表概括恰好生成奇素数，只需证明 $2n+1$ 永远不是合数，也就是说它永远不会分解成 $(2i+1)(2j+1)$，其中 i 和 j 是正整数。为什么？

习题 K 定义函数 $f(\perp) = 0$，如果 $x \neq \perp$，则 $f(x) = 1$，这样定义的函数 f 是可计算的吗？如果定义一个函数对于所有有穷列表和非完整列表返回 $\perp$，对于所有无穷列表返回 1，这样的函数可计算吗？ 233

习题 L 一个圆环（torus）是一个双循环的双向双链表。它在左右方向是循环双链表，而且上下方向也是循环双链表。将矩阵表示为一个长度为 m 的列表，列表的每个元素又都是长度为 n 的列表，构造下列定义：

```
mkTorus :: Matrix a -> Torus a
```

其中：

```
data Torus a = Cell a (Torus a) (Torus a)
                      (Torus a) (Torus a)
elem  (Cell a u d l r) = a
up    (Cell a u d l r) = u
down  (Cell a u d l r) = d
left  (Cell a u d l r) = l
right (Cell a u d l r) = r
```

定义看似复杂，但是答案简短得令人惊奇。

9.8 答案

习题 A 答案 不可以，因为对于任意无穷列表 xs，`foldl1 f xs = undefined`。

习题 B 答案 定义是

```
cycle [] = error "empty list"
cycle xs = ys  where ys = xs ++ ys
```

注意，如果 xs 无穷，那么 `xs ++ ys = xs`，所以 cycle 是无穷列表上的恒等函数。

习题 C 答案 单行定义是

```
fibs :: [Integer]
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

234

习题 D 答案

```
hamming :: [Integer]
hamming = 1: merge (map (2*) hamming)
              (merge (map (3*) hamming)
                     (map (5*) hamming))
```

习题 E 答案 对 n 用归纳法证明 `approx n xs ⊑ xs`。基本情况容易验证，但是归纳情况需要进一步对 xs 做子归纳。子归纳的基情况（空列表和无定义列表）容易验证，归纳步骤如下：

```
  approx (n+1) (x:xs)
=   {定义}
  x:approx n xs
⊑   {(x:)的单调性和归纳假设}
  x:xs.
```

下面命题的证明可以对 xs 归纳。

```
(∀n:approx n xs ⊑ ys) ⇒ xs ⊑ ys
```

对于无定义和空列表是显然的，对于归纳情况，根据 approx 的定义和列表上的逼近序有

```
  (∀n : approx n (x:xs) ⊑ ys)
⇒ xs ⊑ head ys ∧ (∀n : approx n xs ⊑ tail ys)
```

根据归纳假设有

```
x:xs ⊑ head ys:tail ys = ys
```

由此根据极限的定义得到

$$\lim_{n\to\infty} \texttt{approx n xs} = \texttt{xs}$$

习题 F 答案 两个列表 `repeat undefined` 和 `undefined` 不相等，但是

```
(repeat undefined)!!n = undefined!!n
```

对于所有 `n` 成立，因为两边都是 $\perp$。 235

习题 G 答案 必须证明对于所有的自然数 `n` 有下列等式：

```
approx n (iterate f x) = approx n (x:map f (iterate f x))
```

这个断言可以由下列等式推出：

```
approx n (iterate f (f x))
                = approx n (map f (iterate f x))
```

该等式可以对 `n` 归纳证明。对于归纳情况，化简两边。

对于左边的化简：

```
 approx (n+1) (iterate f (f x))
=  {iterate 的定义}
 approx (n+1) (f x:iterate f (f (f x)))
=  {approx 的定义}
 f x: approx n (iterate f (f (f x)))
=  {归纳假设}
 f x: approx n (map f (iterate f (f x)))
```

对于右边的化简：

```
 approx (n+1) (map f (iterate f x))
=  {iterate 和 map 的定义}
 approx (n+1) (f x:map f (iterate f (f x)))
=  {approx 的定义}
 f x: approx n (map f (iterate f (f x)))
```

习题 H 答案 是的，因为

```
foldr xmerge [] (xs:undefined) = xmerge xs undefined
```

而且右边以 `xs` 的第一个元素开始。

习题 I 答案 证明用归纳法。必须首先证明 `crs (n+1)` 是 $c_1 : c_2 : \cdots : c_m : \perp$ 与 p_{n+1} 的倍数无穷列表 $p_{n+1}p_{n+1}$，p_{n+1} $(p_{n+1}+1)$，…归并的结果，其中 $m = p_n^2$。由此给出了直至 p_{n+1}^2 的所有合数。最后，需要证明 $p_{n+2} < p_{n+1}^2$。 236

部分列表 $\texttt{primes}_5$ 生成所有小于 $2209 = 47 \times 47$ 的素数。

习题 J 答案 因为奇整数具有 $2n+1$ 的形式，其中 n 形如 $i+j+2ij$，那么该数被排除在最后列表之外。但是

$$2(i+j+2ij)+1=(2i+1)(2j+1)$$

习题K答案 不是，f不是单调的：$\perp \sqsubseteq 1$，但是$f(\perp) \not\sqsubseteq f(1)$。对于第二个函数（称为$g$），有xs⊑ys意味着$g$(xs)⊑$g$(ys)，所以$g$是单调的。但$g$不是连续的，故不可计算。

习题L答案 定义为

```
mkTorus ass = head (head xss)
 where xss = zipWith5 (zipWith5 Cell)
             ass (rotr xss) (rotl xss)
             (map rotr xss) (map rotl xss)
```

其中rotr和rotl对矩阵的行做旋转，map rotr和map rotl对矩阵的列做旋转。函数zipWith5的定义必须对于后4个参数是严格的。

9.9 注记

Melissa O'Neill撰写了有关生成素数的好文章，参见“The genuine sieve of Eratosthenes”，Journal of Functional Programming 19(1)，95-106，2009。Ben Sijtsma的博士论文Verification and derivation of infinite-list programs（University of Groningen，the Netherlands，1988）研究无穷列表程序的各种特性，而且给出对这种程序进行推理的技术，其中有一章介绍石头-剪刀-布的公平性证明。

我的论文“On building cyclic and shared data structures in Haskell”包含更多关于无穷列表和循环列表应用的例子，参见Formal Aspects of Computing 24(4-6)，609-621，July 2012。也请在下列链接参阅文章“打结”（Tying the knot）

237
~
238

haskell.org/haskellwiki/Tying_the_Knot

Hamming问题在函数式程序设计早期一直用于循环程序的展示。

命令式函数式程序设计

第 2 章描述了函数 putStrLn 是一个 Haskell 命令（command），IO a 是与外界交互的输入 - 输出计算的类型，并返回一个类型 a 的值，还提到一个表示命令顺序执行的语法，称为 do 语法（do-notation）。本章详细解释这些术语的含义，并介绍一种新的程序设计方法，称为单子（monadic）程序设计。单子程序提供了与外界交互的简单且颇具吸引力的方法，而且这种程序的能力远不止于此：它提供了解决一类问题的简单顺序机制，包括异常处理、破坏性数组更新、语法分析和基于状态的计算。单子式程序使得人们能够用函数程序实现诸如 Python 和 C 等命令式程序，具有非常实用的意义。

10.1 IO 单子

类型 IO a 是第 2 章所述的一个抽象数据类型，所以不知道它的称为动作（action）或者命令的值是如何表示的。但是，可以将这个类型想象成

```
type IO a = World -> (a,World)
```

因此，一个动作是一个函数，其输入是一个世界，输出是一个类型 a 的值以及一个新的世界。然后这个新的世界可以作为下一个动作的输入。因为输入 - 输出动作改变着当前世界，所以不能返回到旧世界，也不能重复这个世界或者查看其组成。所能做的只有用给定的原始动作在世界上进行操作，将这些动作用某种顺序组合起来。

一个原始动作是打印一个字符：

```
putChar :: Char -> IO ()
```

当该动作被执行时，一个字符被打印在标准输出上，通常是计算机屏幕。例如：

```
ghci> putChar 'x'
xghci>
```

字符 x 被打印在屏幕上，没有别的事情发生，所以 GHCi 的下一个提示符紧接在 x 后面，没有空格，也没有换行。执行这个动作不产生有价值的值，所以返回值是零元组 ()。

另一个原始动作是 done :: IO ()，该命令不做任何事情，它不改变当前世界，并返回零元组 ()。

顺序组合动作的一个简单运算是 (>>)，其类型是

```
(>>) :: IO () -> IO () -> IO ()
```

给定动作 p 和 q，动作 p >> q 首先完成动作 p，然后完成动作 q。例如：

```
ghci> putChar 'x' >> putChar '\n'
x
ghci>
```

这次在 x 之后打印了换行。利用 (>>) 可以定义函数 putStrLn：

```
putStrLn :: String -> IO ()
putStrLn xs = foldr (>>) done (map putChar xs) >>
              putChar '\n'
```

该动作打印一个串的所有字符，然后用一个换行符结束。注意 map putChar xs 是一个动作列表。我们仍然在函数式程序设计的世界，它的所有表达能力，包括 map 和 foldr 都是可以使用的。

下面是另一个原始动作：

```
getChar :: IO Char
```

这个动作被执行时，从标准输入设备读取一个字符。标准输入设备的输入是用户在键盘上
240 键入的字符，所以 getChar 返回用户键入的第一个字符。例如：

```
ghci> getChar
x
'x'
```

键入 getChat，再按回车后，GHCi 等待用户建入一个字符。键入'x'（键入时该字符同时显示出来），然后该字符被读取，接着打印出来。

done 的推广是不做任何事情，并返回一个给定值的动作：

```
return :: a -> IO a
```

特别是，done = return ()。(>>)的推广具有下列类型：

```
(>>) :: IO a -> IO b -> IO b
```

给定动作 p 和 q，动作 p >> q 先完成动作 p，丢弃该动作返回的值，然后完成动作 q。例如：

```
ghci> return 1 >> return 2
2
```

显然，这个动作只能用于对 p 返回的值不感兴趣的情况，因为 q 没有办法依赖于这个值。人们真正需要的是一个更通用的具有如下类型的运算（>>=）：

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

使用该运算得到的组合 p >>= f 是一个动作：该动作被执行时，首先完成 p，返回类型 a 的值 x，然后完成动作 f x，最后返回一个类型 b 的值 y。很容易用（>>=）定义（>>），将此留作练习。运算（>>=）通常称为绑定（bind），不过也将其读作“然后应用”。

利用（>>=）可以定义一个函数 getLine，用于读取一行输入，更确切地说，读取一个字符列表，直至第一个换行符，但不包括该换行符：

```
getLine :: IO String
getLine = getChar >>= f
          where f x = if x == '\n' then return []
                      else getLine >>= g
                      where g xs = return (x:xs)
```

这个定义的含义很直观：取得第一个字符 x，如果 x 是换行符，则返回空列表；否则取得该
241 行的其他字符，然后将 x 添加在其前面。尽管解读很直接，但是定义中嵌套 where 子句的使用有点难懂。一种改进定义可读性的方法是使用匿名兰姆达表达式写成：

```
getLine = getChar >>= \x ->
          if x == '\n'
          then return []
          else getLine >>= \xs ->
               return (x:xs)
```

另一种更好的方法是使用 do 记法：

```
getLine = do x <- getChar
             if x == '\n'
             then return []
             else do xs <- getLine
                     return (x:xs)
```

定义右边使用了 Haskell 格式惯例。特别要注意条件表达式的缩进，而且最后一个return的缩进表示它属于最里那个 do 的动作。用程序设计者的观点来说，最好使用花括号和分号明确标示格式：

```
getLine = do {x <- getChar;
              if x == '\n'
              then return []
              else do {xs <- getLine;
                       return (x:xs)}}
```

后面还会讨论 do 记法。

Haskell 库 System.IO 提供了 putChar 和 getChar 之外的许多其他动作，包括打开文件和读文件、写文件和关闭文件、各种方式的缓存输出，等等。本书不介绍这些动作的详情。但是，或许有两点需要解释。第一点，不存在类型 IO a -> a 的函数[⊖]。一旦进入完成输入－输出动作的世界，必须呆在这个世界里，不能出来。为了说明必须这样规定的原因，假设有函数 runIO，并考虑下列定义：

```
int :: Int
int = x - y
      where x = runIO readInt
            y = runIO readInt

readInt = do {xs <- getLine; return (read xs :: Int)}
```

242

动作 readInt 读取一行输入，只要输入只含数字，就将其解释为一个整数。那么，现在 int 的值是什么？答案完全取决于 x 和 y 哪个先得到求值。Haskell 没有预先规定在表达式 x - y 中 x 是否在 y 之前先求值。换句话说，输入－输出动作必须用确定的方式顺序进行，而且 Haskell 是惰性函数语言，很难确定其中事情发生的次序。当然，表达式如 x - y 是一个非常简单的例子（在命令式语言中同样不期望的事情也会出现），但是可以想象提供这样的函数 runIO 将带来的混乱。

第二点，或许是应该讲给那些对于如下表达式漫不经心的读者：

```
undefined >> return 0 :: IO Int
```

这个代码引起错误还是返回0？答案是“一个错误”。IO 在下列意义下是*严格的*：IO 动作是按照顺序完成的，即使后续的动作可能不在意前面动作的返回值。

⊖ 实际上存在这样的函数，称为 unsafePerformIO，但它是一个很不安全的函数。

返回本节主题，总结一下。类型 IO a 是一个抽象数据类型，在该类型上至少可以使用下列运算：

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b

putChar :: Char -> IO ()
getChar :: IO Char
```

后两个函数是特定于输入和输出的，但是前两个函数则不然。实际上它们是刻画单子类族的通用顺序运算：

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

两个单子运算需要满足某些定律，将会在适当的时候说明是哪些定律。至于名称“单子”的来源，它是从哲学中偷来的，或者说是从莱布尼兹那里偷来的，而莱布尼兹是从希腊
243 哲学中借来的。请勿深究这个名词。

10.2 更多的单子

如果单子仅仅要求这些，是否一定有很多单子？是的，确实如此。特别是，谦逊的列表类型是一个单子：

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

当然，还不清楚单子运算应该满足什么定律，所以也许这个实例定义不正确（是正确的），但是，至少运算的类型是正确的。因为 do 记法可用于任何单子，所以，可以用新的语法定义函数，如笛卡儿积函数 cp :: [[a]] -> [[a]]（参见 7.3 节）：

```
cp []       = return []
cp (xs:xss) = do {x <- xs;
                  ys <- cp xss;
                  return (x:ys)}
```

比较第二个子句的右边与下列列表概括：

```
[x:ys | x <- xs, ys <- cp xss]
```

可以看出，这两个写法非常相似，唯一的真正区别是在 do 记法中结果出现在最后，而不是在开始。如果在列表概括前引入了单子和 do 记法，也许后者的列表概括就不需要了。

下面是另一个例子。类型 Maybe 是一个单子：

```
instance Monad Maybe where
  return x      = Just x
  Nothing >>= f = Nothing
  Just x >>= f  = f x
```

为了欣赏这个单子带来的便利，考虑 Haskell 库函数：

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

如果(x,y)是列表 alist 中第一个分量为 x 的二元组的首次出现，那么lookup x alist 的值是 Just y；如果不存在这样的二元组，则其值为 Nothing。设想在列表 alist 中查找 x，然后在第二个列表 blist 中查找结果 y，接着继续在第三个列表 clist 中查找 244
结果 z。如果这些查找中任何一个查找返回 Nothing，那么最后结果是 Nothing。定义这个函数，必须用类似下面的式子来表达：

```
case lookup x alist of
  Nothing -> Nothing
  Just y  -> case lookup y blist of
               Nothing -> Nothing
               Just z  -> lookup z clist
```

利用单子，可以如下定义：

```
do {y <- lookup x alist;
    z <- lookup y blist;
         lookup z clist}
```

代之以显式地写出计算链，而且每个计算可能返回 Nothing，并将 Nothing 显式地在计算链中传递，可以把定义写成简单的单子表达式，其中 Nothing 的处理是在单子中隐式地进行的。

do 记法

就像列表概括可以翻译成 map 和 concat 的表达式，do 表达式也可以翻译成用 return和绑定表示的表达式。三个主要翻译规则如下：

```
do {p}            = p
do {p;stmts}      = p >> do {stmts}
do {x <- p;stmts} = p >>= \x -> do {stmts}
```

在这些规则中 p 表示一个动作，所以第一个规则表示单个动作旁的 do 可以去掉。第二个和第三个规则中的 stmts 是一个语句的非空序列，其中每个语句或者是一个动作，或者是一个形如 x <- p 的语句。后者不是一个动作，所以下面的表达式不是语法正确的：

```
do {x <- getChar}
```

顺便说明的是，空的 do 表达式 do { }也不是语法正确的。一个 do 表达式的最后一个语句必须是一个动作。

另一方面，下面两个表达式都是正确的：

```
do {putStrLn "hello "; name <- getLine; putStrLn name}
do {putStrLn "hello "; getLine; putStrLn "there"}
```

245

第一个例子打印一个问候，然后读取一个名，并完成问候。第二个例子打印一个问候，读取一个名，但是即刻忘记该名，然后用“there”结束问候。两个例子有点像生活中一个人被介绍给另一个人的情景。

最后，由以上的翻译规则，可以证明以下两个规则：

```
do {do {stmts}} = do {stmts}
do {stmts1; do {stmts2}} = do {stmts1; stmts2}
```

但是，对于嵌套的 do 必须小心：

```
do {stmts1;
    if p
    then do {stmts2}
    else do {stmts3}}
```

如果 stmts2 和 stmts3 包含不止一个动作，则这里的 do 是必须的。

单子定律

单子定律表述的仅仅是关于 return 和（>>=）的表达式的简化，也正如我们所期待的一样。定律共有三条，我们将用三种不同的方式来叙述它们。第一条定律表示 return 是（>>=）的右单位元：

```
(p >>= return) = p
```

这个定律用 do 记法表示为

```
do {x <- p; return x} = do {p}
```

第二条定律表示 return 也是一种左单位元：

```
(return e >>= f) = f e
```

该定律用 do 记法表示为

```
do {x <- return e; f x} = do {f e}
```

第三条定律表示（>>=）是某种意义上可结合的：

```
((p >>= f) >>= g) = p >>= (\x -> (f x >>= g))
```

246 该定律用 do 记法表示为

```
  do {y <- do {x <- p; f x}; g y}
= do {x <- p; do {y <- f x; g y}}
= do {x <- p; y <- f x; g y}
```

最后一行利用了 do 记法的消去嵌套性质。

对于第三种表示单子定律的方法，考虑如下定义的运算（>=>）：

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(f >=> g) x = f x >>= g
```

这个运算像函数复合，只是复合的函数都具有类型 x -> m y，其中 x 和 y 是某些适当的类型，而且复合的顺序是从左至右，而不是从右向左。该运算（左到右）称为 Kleisli 复合，在 Haskell 库 Control.Monad 中定义。也可以定义对偶的（右到左）Kleisli 复合：

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
```

其定义留作练习。

这里想表达的重点是（>>=）可以用（>=>）定义：

```
(p >>= f) = (id >=> f) p
```

更简短的可表示为(>>=)= flip (id >=>)。另外还有蛙跳（leapfrog）规则：

```
(f >=> g) . h = (f . h) >=> g
```

证明留作练习。

用（>==>）来叙述的单子定律仅仅表示（>==>）有单位元 return，而且满足结合律。一个集合与具有单位元的且满足结合律的二元运算称为一个幺半群（monoid），“单子（monad）”一词可能来自于幺半群的双关语。即使如此，这确实是表述单子定律的最简短方法。

另一种表述单子定律且有指导意义的方法在习题中考虑。

10.3 状态单子

如果不是因为如何解决输入-输出动作的正确顺序进行，或许单子不会出现在 Haskell 中。但是，一旦人们理解了单子的作用，各种单子应用很快接踵而来。已经看到应用 Maybe 单子的运算如何简化涉及在一系列计算中传递备份信息的计算链。单子的另一种基本应用是处理可变（mutable）结构，如数组，出于效率的考虑，需要能够修改其值，因而在这个过程中破坏了它的原始结构。 247

可变结构通过状态线程单子 ST s 引入，该单子将在 10.4 节介绍。在介绍这个单子的性质之前，先考虑一个简单的单子，称为 State s，该单子用于处理显式状态 s。可以将类型 State s a 看作下面的函数：

```
type State s a = s -> (a,s)
```

具有类型 State s a 的动作输入一个初始状态，返回一个类型 a 的值和一个新的状态。但是，想把 IO a 看作 State World a 的同义词是错误的。State s a 中的状态成分 s 可以暴露，也可以被处理，但是不能暴露和处理这个世界。

特别是，除了单子的两个运算 return 和(>>=)之外，状态单子还有另外 5 个运算：

```
put      :: s -> State s ()
get      :: State s s
state    :: (s -> (a,s)) -> State s a
runState :: State s a -> (s -> (a,s))
evalState :: State s a -> s -> a
```

函数 put 将状态置入一个给定的配置中，而 get 返回当前状态。这两个函数都可以用 state 来定义：

```
put s = state (\_ -> ((),s))
get   = state (\s -> (s,s))
```

另一方面，state 也可以用 put 和 get 定义：

```
state f = do {s <- get; let (a,s') = f s;
              put s'; return a}
```

Haskell 允许在 do 表达式中简写 let 表达式（在列表概括中也可简写）。简写规则是

```
do {let decls; stmts} = let decls in do {stmts}
```

函数 runState 是函数 state 的逆：它取得一个动作和一个初始状态，然后返回动作完成后的最终值和最终状态（这是 IO 单子难以完成的）。函数 evalState 定义为 248

```
evalState m s = fst (runState m s)
```

而且只返回某个状态下计算出来的值。

下面是 State 的一个应用例子。在 7.6 节定义了由一个非空列表构造一个二叉树的下列程序：

```
build :: [a] -> BinTree a
build xs = fst (build2 (length xs) xs)
build2 1 xs = (Leaf (head xs),tail xs)
build2 n xs = (Fork u v, xs'')
              where (u,xs')  = build2 m xs
                    (v,xs'') = build2 (n-m) xs'
                    m        = n `div` 2
```

要注意的是，build2 基本上是一个处理类型[a]的状态的函数，并返回 BinTree a 的元素作为结果。另一种定义 build 的方法如下：

```
build xs = evalState (build2 (length xs)) xs

build2 :: Int -> State [a] (BinTree a)
build2 1 = do {x:xs <- get;
               put xs;
               return (Leaf x)}
build2 n = do {u <- build2 m;
               v <- build2 (n-m);
               return (Fork u v)}
           where m = n `div` 2
```

显式处理状态的所有工作是在建立叶结点时完成的。先读取状态，将第一个元素用 Leaf 标识，剩余的列表便作为新的状态。如果说 build2 n 的第一个版本将状态显式地串联起来，那么第二个版本将这些线程隐藏到了单子的面具下面。

注意 build2 的第一行是一个命令 x:xs <- get，其中左边使用了一个模式而不是一个简单变量。如果当前状态碰巧是空列表，那么这个动作失败，并给出一个适当的错误信息。例如：

249

```
ghci>  runState (do {x:xs <- get; return x}) ""
*** Exception: Pattern match failure in do expression ...
```

当然，这种情况对于 build2 1 不会出现，因为该定义只适用于状态为单元素列表的情况。把 build []会出现什么情况留作练习。

另一个例子是生成某个区间的伪随机数问题。设想有如下函数：

```
random :: (Int,Int) -> Seed -> (Int,Seed)
```

该函数输入一对整数作为指定的区间，再输入一个种子，然后计算一个随机数和一个新的种子。新的种子用于获取后面的随机数。与其显式地说明什么是种子，假定下列函数：

```
mkSeed :: Int -> Seed
```

该函数用给定的整数制作一个种子。现在如果想抛一对骰子，可以定义函数：

```
diceRoll :: Int -> (Int,Int)
diceRoll n = (x,y)
             where (x,s1) = random (1,6) (mkSeed n)
                   (y,s2) = random (1,6) s1
```

但是，也可以定义函数：

```
diceRoll n = evalState (
             do {x <- randomS (1,6);
                 y <- randomS (1,6);
                 return (x,y)}
             ) (mkSeed n)
             where randomS = state . random
```

函数 randomS :: (Int,Int) -> State Seed Int 读取一个整数区间，返回一个动作。diceRoll 的第二个定义比第一个稍长一点，但是也更容易书写。设想不是抛两个骰子，而是5个骰子，如大话骰（liar dice）。此时第一个定义方法需要一系列 where 子句表达5个值和5个种子间的串联，很容易写错，但是，第二个定义方法却很容易扩展，又很难写错。

最后一点，考虑：

```
evalState (do {undefined; return 0}) 1
```

这个会引发异常，还是返回0？换言之，单子 State 像单子 IO 是严格的，还是惰性的？答案是两种都可能。状态单子有两种版本，一种是惰性的，另一种是严格的，区别在于如何实现运算（>>=）。Haskell 提供惰性的缺省实现，在 Control.Monad.State.Lazy 中定义，但是用户也可以选择 Control.Monad.State.Strict 中的严格定义。 250

10.4 ST 单子

状态线程单子在库 Control.Monad.ST 中定义，它与状态单子完全不同，尽管两者看上去貌似一样。像 State s a 一样，可以把状态线程单子看作如下类型：

```
type ST s a = s -> (a,s)
```

但其中一个非常重要的区别是，类型变量 s 不能初始化为特定的状态，如 Seed 或者 [Int]，它在这里的作用只是命名状态。把 s 想象成一个标签，用于表示一个特定的状态线程，所有的可变类型都用这个线程标记，因此动作只能在它们自己的状态线程中影响可变值。

一种可变值是程序变量（program varable）。命令式程序语言中的程序变量不同于 Haskell 中的变量，或者数学中的变量。程序变量可以看作其他值的引用，在 Haskell 中是类型 STRef s a 的对象。这里的 s 表示引用局部于状态线程 s（没有其他），这里的 a 是被引用值的类型。在 Data.STRef 中定义了创建引用、读写引用的运算：

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

下面是一个例子。回顾7.6节，给出斐波那契函数的下列定义：

```
fib :: Int -> Integer
fib n  = fst (fib2 n)
fib2 0 = (0,1)
fib2 n = (b,a+b)  where (a,b) = fib2 (n-1)
```

251

对 fib 求值运行时间是线性的，但是所使用的空间不是常数（即使忽略任意大的整数不能在常数空间存储的事实）：每个递归调用都涉及新变量 a 和 b。对比之下，下面是命令

式语言 Python 中 fib 的定义：

```
def fib (n):
    a,b = 0,1
    for i in range (0,n):
        a,b = b,a+b
    return a
```

定义处理两个程序变量 a 和 b，运行空间是常数（至少对于小整数）。几乎可以直接把Python代码翻译成 Haskell 代码：

```
fibST :: Int -> ST s Integer
fibST n = do {a <- newSTRef 0;
              b <- newSTRef 1;
              repeatFor n
               (do {x <- readSTRef a;
                    y <- readSTRef b;
                    writeSTRef a y;
                    writeSTRef b $! (x+y)});
              readSTRef a}
```

注意严格函数应用运算符（$）的使用强制对和求值。动作 repeatFor 重复一个动作给定的次数：

```
repeatFor :: Monad m => Int -> m a -> m ()
repeatFor n = foldr (>>) done . replicate n
```

一切工作正常，但是最后得到的是一个动作 ST s Integer，而我们想要的是一个整数。如何脱离单子回到 Haskell 值的世界？答案是提供一个类似于状态单子的 runState 的函数，下面是它的类型：

```
runST :: (forall s. ST s a) -> a
```

这个类型不同于以前看到的其他 Haskell 类型，它被称为二阶多态类型（rank 2 polymorphic type），过去所有的多态类型是一阶的。这里表达的是，runST 的参数对 s 必须是一致的，所以它不能依赖于 s 名称之外的任何信息。特别是，动作中每个 STRef 的声明必须带上同一个线程名 s。

252

为了进一步说明二阶类型，考虑下列两种列表之间的区别：

```
list1 :: forall a. [a -> a]
list2 :: [forall a. a -> a]
```

类型 list1 只是我们熟悉的类型[a -> a]，因为对一阶类型假定了在最外围有全称量词对类型变量的约束。例如，将 list1 的类型变量 a 初始化为类型 Float，则[sin,cos,tan]是 list1 的可能值。但是，只有两个函数可以成为 list2 的元素，即 id 和无定义函数 undefine，因为只有这两个函数具有类型 forall a. a -> a。如果给出类型 a 的一个元素 x，但对类型 a 没有任何其他信息，那么返回类型 a 的一个值只有两种选择，x 或者⊥。

为什么 runST 需要二阶类型呢？嗯，这样可以避免定义下面的式子：

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```

这样的代码不是类型正确的，因为

```
newSTRef True :: ST s (STref s Bool)
```

而且在表达式 runST (newSTRef True)中，Haskell 类型检测器不能将 STRef s a 与 runST 期望的结果类型 a 匹配。类型 STRef s a 的值不能从 ST a 输出，只有不依赖于 s 的类型的对象可以输出。如果运行这样的代码，那么在第一个 runST 中分配的引用在第二个 runST 内也可以使用。这样就使得一个线程的读也可用于另一个线程，因此结果依赖于执行这些线程的次序，从而导致混乱和困惑。这也正是在 IO 单子中极力避免发生的问题。

但是，可以安全地定义：

```
fib :: Int -> Integer
fib n = runST (fibST n)
```

这个 fib 定义的运行时间是线性的。

对用户来说，ST 单子的主要应用在于它处理可变数组的能力。数组的整个问题值得用一节讨论。 253

10.5 可变数组

函数式程序设计中重点的基本数据结构是列表，而不是数组，这点有时让第一次遇到函数式程序设计的命令式程序员感到吃惊。原因是数组的大多数使用（尽管不是全部）效率依赖于它们的修改是破坏性的。一旦修改了数组在某个下标处的值，那么旧数组便丢失了。但是，在函数式程序设计中，数据结构是持久的（persistent），而且任何命名的结构会持续存在。例如，insert x t 可能在树 t 中插入一个元素 x，但是 t 仍然表示原树，所以最好不要覆盖。

在 Haskell 中一个可变数组是类型 STArray s i e 的对象。这里 s 表示状态线程，i 表示索引类型，e 是元素类型。并非任何类型都可以做索引类型，合法的索引是类族 Ix 的成员。这个类族的成员包括 Int 和 Char，能够映射到整数的某个连续区域的对象。

像 STRefs 一样，存在创建、读和写数组的运算。下面将用一个简洁的例子说明这些运算。回顾 7.7 节的快速排序：

```
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++
               qsort [y | y <- xs, x <= y]
```

当时就说，快速排序是用数组实现的，而不是用列表；划分可以原地（in place）完成，而不需要额外空间。现在具备了实现这种算法的工具。首先给出如下定义：

```
qsort :: (Ord a) => [a] -> [a]
qsort xs = runST $
           do {xa <- newListArray (0,n-1) xs;
               qsortST xa (0,n);
               getElems xa}
           where n = length xs
```

首先创建一个具有边界(0,n-1)的可变数组，并用 xs 的元素填充；然后数组上的排序由动作 qsortST xs (0,n)完成；最后返回有序数组元素构成的列表。上面代码中，

动作 newListArray 具有类型：

254
```
Ix i => (i, i) -> [e] -> ST s (STArray s i e)
```

而且 getElems 具有类型：

```
Ix i => STArray s i e -> ST s [e]
```

第一个动作用一个列表构造一个可变数组，第二个返回一个可变数组中元素的列表。

qsortST xa (a,b)的目标是将 xa 在区间(a,b)上的子数组排序，根据定义这个区间包含下界，但不包含上界，换句话说是区间[a .. b-1]。选择左闭右开区间几乎总是处理数组的最好选择。下面是 qsortST 的定义：

```
qsortST :: Ord a => STArray s Int a ->
           (Int,Int) -> ST s ()
qsortST xa (a,b)
  | a == b    = return ()
  | otherwise = do {m <- partition xa (a,b);
                    qsortST xa (a,m);
                    qsortST xa (m+1,b)}
```

如果 a==b，则区间为空，无需做任何工作；否则，将数组元素重写排列，使得对于某个适当的 x，在区间(a,m)的所有元素都小于 x，在区间(m+1,b)上的所有元素都大于或者等于 x，元素 x 本身存放在位置 m 处，然后对两个子区间排序，从而结束排序。

接下来的工作是定义 partition。寻找合适定义的唯一方法是利用前置条件、后置条件和循环不变量进行形式化的开发。但是，本书是关于函数式程序设计的，不是命令式程序的形式化开发，所以这里仅仅给出一种定义：

```
partition xa (a,b)
 = do {x <- readArray xa a;
       let loop (j,k)
            = if j==k
              then do {swap xa a (k-1);
                       return (k-1)}
              else do {y <- readArray xa j;
                       if y < x then loop (j+1,k)
                       else do {swap xa j (k-1);
                                loop (j,k-1)}}
255
       in loop (a+1,b)}
```

动作 swap 的定义如下：

```
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap xa i j =  do {v <- readArray xa i;
                   w <- readArray xa j;
                   writeArray xa i w;
                   writeArray xa j v}
```

下面解释 partition 是如何进行划分的，这种解释简洁但一定不够充分。用第一个元素作为枢纽，然后进入循环处理剩余的区间(a+1,b)，直至区间为空时停止。跳过小于 x 的元素，区间左边界右移。当遇到大于 x 的元素 y 时，将其与区间最右元素交换，右边界左移。当区间为空时，将枢纽放在它的最终位置上，并将这个位置作为结果返回。

注意 loop 是单子中的一个局部过程，也可以将其定义为一个全局过程，这样便需要增加 3 个参数，即数组 xa、枢纽 x 和开始位置 a。

哈希表

纯函数式快速排序与基于可变数组的快速排序具有相同的渐进时间性能，但是在个别场合可变数组似乎对于获得渐进更快的算法起到关键的作用。一种情况是哈希表中集合的有效表示。

下面用一个特殊问题说明哈希表的应用。考虑用两个有限集定义的典型谜题，一个位置（position）集合和一个移动（move）集合。以下是给定的函数：

```
moves  :: Position -> [Move]
move   :: Position -> Move -> Position
solved :: Position -> Bool
```

函数 moves 描述在一个给定位置可以进行移动的集合，solved 确定构成谜题解的位置。解谜题就是找到一个从给定开始位置到解位置的移动序列，最好是最短序列： 256

```
solve :: Position -> Maybe [Move]
```

如果不存在从位置 p 到一个解位置的移动序列，那么值 solve p 是 Nothing，否则是 Just ms，满足：

```
solved (foldl move p ms)
```

下面将通过宽度优先（breadth first）搜索实现 solve。宽度优先搜索指先检查从开始位置一次移动能够到达的所有位置，然后检查两次移动能到达的所有位置，等等。因此，如果存在解，那么宽度优先搜索将找到最短解。为实现宽度优先搜索，需要下列类型：

```
type Path     = ([Move],Position)
type Frontier = [Path]
```

一条路径由一个从开始位置的移动序列（按照逆序表示）和这些移动到达的最后位置构成。前沿（frontier）是一个有待扩展成更长路径的列表。宽度优先搜索于是可以如下定义：

```
solve p = bfs [] [([],p)]

bfs :: [Position] -> Frontier -> Maybe [Move]
bfs ps [] = Nothing
bfs ps ((ms,p):mps)
 | solved p    = Just (reverse ms)
 | p `elem` ps = bfs ps mps
 | otherwise   = bfs (p:ps) (mps ++ succs (ms,p))

succs :: Path -> [Path]
succs (ms,p) = [(m:ms,move p m) | m <- moves p]
```

函数 bfs 的第一个参数 ps 表示已经访问位置的集合。第二个参数是前沿，并使用队列处理，以保证相同长度的路径处理后再处理它们的扩展。检查一个路径时；如果其最后位置是解位置，则接受该路径；如果其最后位置已经访问过，则拒绝该路径，否则在前沿后面添加后继移动扩展路径。成功路径的移动序列在作为结果返回前先置逆，这是为了 succs 在添加后继时可以将后继加在列表的前面，而不是加在列表尾部，以提高效率。

函数 bfs 中影响效率的主要来源有两个，一个是（++）的使用，另一个有关 elem。 257
首先，前沿的规模可能呈指数式增长，所以在前沿的尾部添加后继很慢。最好的方法是如下定义 bfs：

```
bfs :: [Position] -> Frontier -> Frontier ->
       Maybe [Move]
bfs ps [] []  = Nothing
bfs ps [] mqs = bfs ps mqs []
bfs ps ((ms,p):mps) mqs
 | solved p    = Just (reverse ms)
 | p `elem` ps = bfs ps mps mqs
 | otherwise   = bfs (p:ps) mps (succs (ms,p) ++ mqs)
```

附加的参数是临时前沿，用于存储后继。当第一个前沿已耗尽为空时，临时前沿成为新的前沿。在临时前沿前面添加后继需要的时间正比于后继的个数，而不是前沿的规模，所以算法运行更快。另一方面，新的 bfs 不同于旧的定义，因为后续的前沿交替地从左到右与从右到左遍历。不过，只要存在解，那么最短解仍将被找到。

第二个低效来源是成员测试。用列表存储已经访问过的位置效率低，因为成员测试花费的时间可能正比于当前访问过的位置数。如果位置用区间 [0..n-1] 的整数表示，那么问题会好得多，因为此时可以用边界为 (0,n-1) 的布尔数组标记访问过的位置，成员测试只需要在单个数组上进行查找。

可以设想将位置用整数编码，但并不是用自然数的某个初始段。例如，数独位置（见第5章）可以表示成81位整数。所以，假设有将位置用整数编码的函数：

```
encode :: Position -> Integer
```

为了缩小表示区间，对于某个适当的 n :: Int，定义：

```
hash :: Position -> Int
hash p = fromInteger (encode p) `mod` n
```

函数 hash 的结果是区间 [0..n-1] 中的一个整数。

258 一个问题，也是最大的问题：不同的位置可能哈希到同一个整数。解决这个问题时，我们不使用布尔数组，而使用位置列表数组。数组下标 k 处存储哈希值为 k 的列表。虽然并不能保证在最坏情况下能够改进算法效率，但是，如果 n 取适当大的整数，而且哈希函数相对均匀地将整数赋给位置，那么成员测试的复杂度可以减小为原来的 $1/n$。

使用哈希方法，修改后的 solve 定义如下：

```
solve :: Maybe [Move]
solve = runST $
        do {pa <- newArray (0,n-1) [];
            bfs pa [([],start)] []}

bfs :: STArray s Int [Position] -> Frontier ->
       Frontier -> ST s (Maybe [Move])
bfs pa [] []  = return Nothing
bfs pa [] mqs = bfs pa mqs []
bfs pa ((ms,p):mps) mqs
 = if solved p then return (Just (reverse ms))
   else do {ps <- readArray pa k;
            if p `elem` ps
            then bfs pa mps mqs
            else
            do {writeArray pa k (p:ps);
                bfs pa mps (succs (ms,p) ++ mqs)}}
   where k = hash p
```

10.6 不变数组

讲到数组，必须指出 Haskell 提供了一个很好的库 Data.Array，该库提供了不变数组上的纯函数式运算。运算用可变数组实现，但接口是纯函数式的。

类型 Array i e 是数组的抽象类型，其中 i 是下标类型，e 是元素类型。构造数组的基本运算是

```
array :: Ix i => (i,i) -> [(i,e)] -> Array i e
```

259

函数的第一个输入是一对边界，即数组的最小下标和最大下标；第二个参数是下标和元素对的联合列表，说明数组在各个下标处的元素；结果是给定边界和元素的数组。在联合列表中没有说明元素值的数组元素将是无定义值。如果两个元素有相同的下标，或者其中一个下标越界，则结果是无定义的数组。因为有了这些测试，数组的构造对于下标是严格的，但是对于元素则是惰性的。构造数组的时间复杂度与元素个数呈线性关系。

一个简单的 array 特例是 listArray，其输入中只需说明元素列表：

```
listArray :: Ix i => (i,i) -> [e] -> Array i e
listArray (l,r) xs = array (l,r) (zip [l..r] xs)
```

另外一种构造数组的方法称为 accumArray，其类型看上去有些吓人：

```
Ix i => (e -> v -> e) -> e -> (i,i) -> [(i,v)] -> Array i e
```

第一个参数是“累积函数”，用于将数组元素和新值转换为新的数组元素；第二个参数是每个数组元素的初始值；第三个参数是一对上下界；第四个也就是最后一个参数是下标和值的联合列表；结果是一个数组。构造方法是将累积函数应用于初始值与联合列表的每个元素，得到数组相应位置的新元素。如果假定累积函数是常数时间的，那么构造过程的运行时间线性于联合列表的长度。

以上是用语言描述的 accumArray 的功能，用符号表示如下：

```
elems (accumArray f e (l,r) ivs)
  = [foldl f e [v | (i,v) <- ivs, i==j] | j <- [l..r]]
```

其中 elems 按照数组下标顺序返回数组的元素列表。不过，以上等式不完全正确：对于 ivs 需要添加约束条件，即每个下标应该介于指定的区间。如果这个条件不满足，那么左边返回错误，而右边则不然。

函数 accumArray 看似复杂，但它是解决某类问题的一个非常有用的工具。这里给出两个例子。第一，考虑有向图的表示。有向图往往在数学上描述为一个结点（vertex）集和一个边（edge）集。一条边是一个结点有序对（j，k），表示该边由结点 j 指向结点 k，称结点 k 邻接于结点 j。以下假设用 1 到 n 的整数表示结点，n 是某个整数。因此，有

```
type Vertex = Int
type Edge   = (Vertex,Vertex)
type Graph  = ([Vertex],[Edge])

vertices g = fst g
edges g    = snd g
```

260

在程序设计中，有向图经常用邻接表表示：

```
adjs :: Graph -> Vertex -> [Vertex]
adjs g v = [k | (j,k) <- edges g, j==v]
```

这种 adjs 定义的问题在于计算任何特定结点的邻接表需要的时间与边数成正比。更好的方法是用数组实现 adjs：

```
adjArray :: Graph -> Array Vertex [Vertex]
```

因此有

```
adjs g v = (adjArray g)!v
```

其中（!）表示数组下标索引运算。对于相对合理规模的数组，该运算是常数时间的。

下面是 adjArray 的说明：

```
elems (adjArray g)
 = [[k | (j,k) <- edges g, j==v] | v <- vertices g]
```

利用这个说明可以直接计算 adjArray 的定义。为了缩短代码行，将 edges g 简记为 es，vertices g 简记为 vs，则有

```
elems (adjArray g) = [[k | (j,k) <- es, j==v] | v <- vs]
```

关注等式右边，第一步用定律 foldr (:) [] = id 重写。由此给出下列表达式：

```
[foldr (:) [] [k | (j,k) <- es, j==v] | v <- vs]
```

下一步利用定律 foldr f e xs = foldl (flip f) e (reverse xs)，其中 xs 是任意有穷列表。简写 flip (:)为(@)，得到

```
[foldl (@) [] (reverse [k | (j,k) <- es, j==v]) | v <- vs]
```

将 reverse 分配后得到表达式：

```
[foldl (@) [] [k | (j,k) <- reverse es, j==v] | v <- vs]
```

261 下一步利用 swap (j,k) = (k,j)得到

```
[foldl (@) [] [j | (k,j) <- es', j==v] | v <- vs]
```

其中 es' = map swap (reverse es)。最后，利用 n = lengh vs 和 accumArray 的说明得到

```
  elems (adjArray g)
= elems (accumArray (flip (:)) [] (1,n) es')
```

这表明可以定义：

```
adjArray g = accumArray (flip (:)) [] (1,n) es
             where n  = length (vertices g)
                   es = map swap (reverse (edges g))
```

这个 adjArray g 的定义可以用正比于边数的时间计算后继。

下面是 accumArray 的第二个应用例子。假设给定 n 个整数的列表，所有整数在区间 $(0, m)$ 中，m 是某个整数。可以通过计算每个元素出现的次数，用 $\Theta(m+n)$ 步对

这个列表排序：

```
count :: [Int] -> Array Int Int
count xs = accumArray (+) 0 (0,m) (zip xs (repeat 1))
```

其中 repeat 1 是无穷个 1 的列表。计数需要 $\Theta(n)$ 步。计数完成后，可以如下排序：

```
sort xs = concat [replicate c x
                  | (x,c) <- assocs (count xa)]
```

函数 assocs 也是一个库函数，它返回一个数组中所有下标和元素对构成的列表，并按照下标顺序排列，排序可以在 $\Theta(m)$ 步完成。

除以上几个运算外，库 Data.Array 还包含一两个其他运算，包括更新运算（//）：

```
(//) :: Ix i => Array i e -> [(i,e)] -> Array i e
```

例如，如果 xa 是 $n \times n$ 矩阵，那么

```
xa // [((i,i),0) | i <- [1..n]]
```

表示同一个矩阵，只是对角线元素都变成了 0。运算（//）的缺点是它的运行时间与数组的长度成正比，即便只更新一个元素也如此。原因是旧的数组 xa 依旧存在，所以必须构建一个全新的数组。

在本章结尾又回到了纯函数式程序设计世界，在这个世界中等式推理既可以用来计算定义，也可以优化定义。尽管单子式程序对于习惯于命令式程序设计的程序员很有吸引力，但是对于如何对单子式程序进行推理是一个问题。当然，等式推理可应用于一定的场合（见习题 F 的例子），但是等式推理不能像在纯函数式程序设计中那么广泛应用（快速排序的正确性见证了这点）。命令式程序员也有同样的问题，他们的解决方法（如果他们愿意这样做的话）是利用谓词演算、前置条件、后置条件和循环不变量。如何对单子式代码进行推理仍然是当前的研究课题。 262

笔者的建议是，保守使用单子式程序，而且只有确实有用时使用；否则，函数式程序设计最重要的特色，即对代码进行数学推理的能力便失去了。

10.7 习题

习题 A 回顾定义：

```
putStr = foldr (>>) done . map putChar
```

请问下列代码的功能是什么？

```
foldl (>>) done . map putChar
```

请将（>>）用（>>=）代替，然后用单子律证明得到的结论。

习题 B 使用模式匹配的方法定义下面的函数：

```
add3 :: Maybe Int -> Maybe Int -> Maybe Int -> Maybe Int
```

该函数将 3 个数相加，如果存在的话

现在用单子 Maybe 重写定义 add3。

习题 C 10.1 节中 cp 的定义仍然是低效的。更好的方法可能是定义： 263

```
cp (xs:xss) = do {ys <- cp xss;
                  x <- xs;
                  return (x:ys)}
```

根据定义，一个单子是可交换的，如果下面等式成立：

```
  do {x <- p; y <- q; f x y}
= do {y <- q; x <- p; f x y}
```

IO 单子显然不是可交换的，然而有的单子是可交换的。请问单子 Maybe 是可交换的吗？

习题 D 每个单子是一个函子。请完成下列定义：

```
instance Monad m => Functor m where
  fmap :: (a -> b) -> m a -> m b
  fmap f = ...
```

目前 Haskell 不要求类族 Monad 必须是类族 Functor 的子类族，但是在将来发行的版本中有改变这种情况的计划。目前 Haskell 为单子提供了一个等价于 fmap 的函数 liftM。请用 return 和 >>= 给出 liftM 的定义。

函数 join :: m (m a) -> m a 将双层单子结构扁平化为单层。请用 >>= 定义 join。对于列表单子，join 和 liftM 给出哪些熟悉的函数？

最后，使用 join 和 liftM 定义（>>=）。由此得出，定义单子时也可以不用 return和 >>=，而使用 return、liftM 和 join。

习题 E 几个有用的单子函数定义在库 Control. Monad 中。例如：

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>)  done
```

（在 Haskell 中有些地方习惯用下划线表示动作的结果是单位元类型。）定义相关的函数：

```
sequence :: Monad m => [m a] -> m [a]
```

264 请利用这两个函数定义下列函数：

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
```

另外，请定义：

```
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
```

在本书中曾用到函数 repeatFor n 重复一个动作 n 次。请将该函数推广为下列函数：

```
for_ :: Monad m => [a] -> (a -> m b) -> m ()
```

习题 F 下面是单子式等式推理的练习。考虑函数：

```
add :: Int -> State Int ()
add n = do {m <- get; put (m+n)}
```

任务是证明：

```
sequence_ . map add = add . sum
```

其中 sequence_是习题 E 中定义的函数，sum 求一个整数列表的累加和。证明需要

foldr 的融合律、put 和 get 的简单定律，以及单子律：

```
do {stmts1} >> do {stmts2} = do {stmts1;stmts2}
```

这个定律成立的条件是 stmts1 和 stmts2 不相交。

习题 G 证明蛙跳规则：(f >=> g) . h = (f . h) >=> g。然后使用这条规则证明(return . h) >=> g = g . h。

习题 H 请证明：

```
liftM f = id >=> (return . f)
join    = id >=> id
```

描述单子律的第四种方法是使用习题 D 的两个函数 liftM 和 join。关于这两个函数有 7 条定律，每一条看上去都眼熟：

265

```
liftM id      = id
liftM (f . g) = liftM f . liftM g

liftM f . return = return . f
liftM f . join   = join . liftM (liftM f)

join . return       = id
join . liftM return = id
join . liftM join   = join . join
```

请证明第四条定律。

习题 I 请问 build []的作用是什么（见 10.3 节）？

习题 J 编写一个玩猜单词游戏 hangman 的交互程序。一个交互过程如下：

```
ghci> hangman
I am thinking of a word:
-----
Try and guess it.
guess: break
-a---
guess: parties
Wrong number of letters!
guess: party
-appy
guess: happy
You got it!
Play again? (yes or no)
no
Bye!
```

假设秘密词的列表存储在名为 Words 的文件中，所以，动作 xs <- readFile "Words"将文件作为一个字符列表读出。另外，readFile 是惰性的，即它根据需要读取内容。

习题 K 请使用一次 STRef 的一个 fibST 定义 fib 的另一个版本。

266

习题 L 定义两个正整数的最大公约数的一种方法如下：

```
gcd (x,y) | x==y = x
          | x<y  = gcd (x,y-x)
          | x>y  = gcd (x-y,y)
```

将该定义翻译成两个程序，一个使用单子 State，另一个使用单子 ST。

习题 M 这里是一个具体的谜题，可以用宽度优先搜索来解决。Sam Loyd 的著名 15 数字推盘游戏的一个简单版本是 8 数字推盘游戏。给出一个 3×3 的阵列，包含了标有数字 1～8 的方块，另有一个空位。可以将一个方块推入临近的空位。根据空位所在位置，可以将标有数字的方块上移、下移、左移或者右移。游戏开始时空位在左上角，其他方块按照 1～8 有序排列。游戏结束时，空位在右下角，其他方块依然按照 1～8 有序排列。

如果愿意接受挑战的话，任务是给出位置和移动的合适表示，并定义函数 moves、move、solved 和 encode。

10.8 答案

习题 A 答案 断言(>>) :: IO () -> IO () -> IO ()满足结合律，并有单位元 done。这表明对于所有有穷串 xs 有

```
putStr xs = foldl (>>) done (map putChar xs)
```

下面重点给出结合律的证明。首先，对于 IO ()中的动作有

267

```
p >> q = p >>= const q
```

其中 const x y = x。现在可以做如下推理：

```
  (p >> q) >> r
=  {(>>)的定义}
  (p >>= const q) >>= const r
=  {第三个单子律}
  p >>= const (q >>= const r)
=  {(>>)的定义}
  p >>= const (q >> r)
=  {(>>)的定义}
  p >> (q >> r)
```

习题 B 答案 直接的定义使用了有通配符的模式匹配：

```
add3 Nothing _ _                = Nothing
add3 (Just x) Nothing _         = Nothing
add3 (Just x) (Just y) Nothing  = Nothing
add3 (Just x) (Just y) (Just z) = Just (x+y+z)
```

这个定义保证 add Nothing undefined = Nothing。

单子式定义如下：

```
add3 mx my mz
 = do {x <- mx; y <- my; z <- mz;
       return (x + y + z)}
```

习题 C 答案 是的。交换律如下：

```
  p >>= \x -> q >>= \y -> f x y
= q >>= \y -> p >>= \x -> f x y
```

对于单子 Maybe，需要验证 4 种可能的情况。例如，如果 p = Nothing, q = Just y,

那么两边都简化为 Nothing。其他情况的验证类似。

习题 D 答案 定义为 268

```
fmap f p = p >>= (return . f)
join p   = p >>= id
```

对于列表单子，有 liftM = map 和 join = concat。

另一个方向的定义为

```
p >>= f = join (liftM f p)
```

习题 E 答案 函数 sequence 的定义为

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr k (return [])
 where k p q = do {x <- p; xs <- q; return (x:xs)}
```

两个新的映射函数定为

```
mapM_ f = sequence_ . map f
mapM f  = sequence . map f
```

函数 foldM 定义为

```
foldM :: Monad m => (b -> a -> m b) ->
        b -> [a] -> m b
foldM f e []     = return e
foldM f e (x:xs) = do {y <- f e x; foldM f y xs}
```

注意 foldM 类似于 foldl，从左到右进行。最后，for = flip mapM_。

习题 F 答案 首先注意到，利用 6.3 节给出的 foldr 和 map 的融合律得到

```
  sequence_ . map add
= foldr (>>) done . map add
= foldr ((>>) . add) done
```

此外，有

```
((>>) . add) n p = add n >> p
```

因为 sum = foldr (+) 0，这表明我们必须证明：

```
foldr (\ n p -> add n >> p) = add . foldr (+) 0
```

269

这个式子看上去像 foldr 融合律的一种特例。因此，需要证明 add 是严格的（是的），而且有

```
add 0 = done
add (n + n') = add n >> add n'
```

下面是证明：

```
  add 0
=  {定义}
  do {m <- get; put (m+0)}
=  {算术}
  do {m <- get; put m}
=  {put和get的简单定律}
  done
```

由此结束第一个等式的证明。对于第二个等式，从较复杂一边开始推理：

```
  add n >> add n'
=  {定义}
  do {l <- get; put (l + n) } >>
  do {m <- get; put (m + n')}
=  {单子律}
  do {l <- get; put (l + n); m <- get; put (m + n')}
=  {put和get的简单定律}
  do {l <- get; put ((l + n) + n')}
=  {(+)的结合律；add的定义}
  add (n + n')
```

习题 G 答案 推理如下：

```
  (f >=> g) (h x)
=  {(>=>)的定义}
  f (h x) >>= g
=  {(>=>)的定义}
  (f . h >=> g) x
```

270

对于第二部分：

```
  (return . h) >=> g
=  {蛙跳规则}
  (return >=> g) . h
=  {单子律}
  g . h
```

习题 H 答案 简化第四条规则的两边。对于左边的化简：

```
  liftM f . join
=  {定义}
  (id >=> (return . f)) . (id >=> id)
=  {蛙跳规则及id . f = f}
  (id >=> id) >=> (return . f)
```

对于右边的化简：

```
  join . liftM (liftM f)
=  {定义}
  (id >=> id) . (id >=> return . (id >=> (return . f)))
=  {蛙跳规则及(>=>)的结合律}
  id >=> (return . (id >=> (return . f))) >=> id
=  {因为 (return . h) >=> g = g . h}
  id >=> id >=> (return . f)
```

因为（>=>）满足结合律，所以两边相等。

习题 I 答案 Build []引起无穷循环，所以其值为⊥。

习题 J 答案 主函数定义如下：

```
hangman :: IO ()
hangman = do {xs <- readFile "Words";
              play (words xs)}
```

271

函数 play 对文件中的不同词玩任意多次（假定总是有足够的词）：

```
play (w:ws)
= do {putStrLn "I am thinking of a word:";
      putStrLn (replicate (length w) '-');
      putStrLn "Try and guess it.";
      guess w ws}
```

函数 guess 处理一次猜测，并将剩余的词留作后续的游戏：

```
guess w ws
= do {putStr "guess: ";
      w' <- getLine;
      if length w' /= length w then
      do {putStrLn "Wrong number of letters!";
          guess w ws}
      else if w' == w
      then
      do {putStrLn "You got it!";
          putStrLn "Play again? (yes or no)";
          ans <- getLine;
          if ans == "yes"
          then play ws
          else putStrLn "Bye!"}
      else do {putStrLn (match w' w);
               guess w ws}}
```

最后定义 match：

```
match w' w = map check w
 where
 check x = if x `elem` w' then x else '-'
```

习题 K 答案 下面程序正确，但是运行空间不是常数空间：

```
fib n = fst $ runST (fibST n)

fibST :: Int -> ST s (Integer,Integer)
fibST n = do {ab <- newSTRef (0,1);
repeatFor n
 (do {(a,b) <- readSTRef ab;
      writeSTRef ab $! (b,a+b)});
readSTRef ab}
```

272

原因是(b,a+b)已经是首范式，所以严格应用不起作用。为了迫使分量的运算，倒数第二行需要修改为

```
b `seq` (a+b) `seq` writeSTRef ab (b,a+b)
```

习题 L 答案 使用单子 State 的定义：

```
gcd (x,y) = fst $ runState loop (x,y)

loop :: State (Int,Int) Int
loop = do {(x,y) <- get;
           if x == y
           then return x
           else if x < y
           then do {put (x,y-x); loop}
           else do {put (x-y,y); loop}}
```

使用单子 ST 的定义：

```
gcd (x,y) = runST $
            do {a <- newSTRef x;
                b <- newSTRef y;
                loop a b}

loop :: STRef s Int -> STRef s Int -> ST s Int
loop a b
 = do {x <- readSTRef a;
       y <- readSTRef b;
       if x==y
       then return x
       else if x<y
       then do {writeSTRef b (y-x);loop a b}
       else do {writeSTRef a (x-y);loop a b}}
```

273

习题 M 答案 当然有许多答案。下面选择的是用9个数字的列表[0..8]表示方块的阵列，其中0表示空位。为了避免重复计算，一个位置用一个二元组（j，ks）表示，其中j是空位在ks中的位置，ks是[0..8]的置换。因此，有

```
type Position = (Int,[Int])
data Move     = Up | Down | Left | Right

encode :: Position -> Integer
encode (j,ks) = foldl op 0 ks
  where op x d = 10*x + fromIntegral d

start :: Position
start = (0,[0..8])
```

函数 moves 可以定义为

```
moves :: Position -> [Move]
moves (j,ks)
  = [Up    | j `notElem` [6,7,8]] ++
    [Down  | j `notElem` [0,1,2]] ++
    [Left  | j `notElem` [2,5,8]] ++
    [Right | j `notElem` [0,3,6]]
```

定义说明，如果空位不在最底行，则允许上移；如果空位不在最顶行，则可以下移；如果空位不在最右列，则可以左移；如果空位不在最左列，则可以右移。

函数 move 可以如下定义：

```
move :: Position -> Move -> Position
move (j,ks) Up    = (j+3,swap (j,j+3) ks)
move (j,ks) Down  = (j-3,swap (j-3,j) ks)
move (j,ks) Left  = (j+1,swap (j,j+1) ks)
move (j,ks) Right = (j-1,swap (j-1,j) ks)

swap (j,k) ks = ks1 ++ y:ks3 ++ x:ks4
 where (ks1,x:ks2) = splitAt j ks
       (ks3,y:ks4) = splitAt (k-j-1) ks2
```

274

最后定义：

```
solved :: Position -> Bool
solved p = p == (8,[1,2,3,4,5,6,7,8,0])
```

我的计算机生成如下结果：

```
ghci> solve start
Just [Left,Up,Right,Up,Left,Left,Down,
      Right,Right,Up,Left,Down,Down,Left,
      Up,Up,Right,Right,Down,Left,Left,Up]
(4.84 secs, 599740496 bytes)
```

10.9 注记

请阅读《Haskell 的历史》（The History of Haskell）了解单子如何成为 Haskell 不可分割的组成部分，以及为什么单子思想对于 Haskell 实际应用的扩展意义重大。编译器的每个阶段都使用单子记录信息。例如，类型检测器使用单子结合状态（维护当前代换）、名称供应（新的类型变量名）和异常。

do 记法优先于（>=>）的使用是由 John Launchbury 在 1993 年提出的建议，并由 Mark Jones 第一次在 Gofer 中实现。

过去几年关于单子的介绍资料不断增加，下面链接给出比较全面的有关文章列表：

haskell. org/haskellwiki/Monad_tutorials

单子等式推理的例子（习题 F）参见 Jeremy Gibbons 的论文“Unifying theories of programming with monads”（UTP Symposium，August 2012）。更多有关单子等式推理的资料参见 Jeremy Gibbons 和 Ralf Hinze 的论文“Just do it：simple monadic equational reasoning”，该论文刊登在 2011 年函数式程序设计国际会议（2011 International Conference of Functional Programming）论文集中。两篇论文均可在下列链接找到：

www. cs. ox. ac. uk/people/jeremy. gibbons/publications/ 275

第11章
Thinking Functionally with Haskell

句法分析

一个句法分析器（parser）是对一个文本进行分析以确定其逻辑结构的函数。文本是描述人们感兴趣值的一个字符串，如算术表达式、一首诗或者一个数据表。句法分析器的结果是对该值的一种表示，如算术表达式的分析结果是某种树，一首诗的分析结果是诗行的列表，数据表的分析结果可能是更复杂的表示。许多程序设计任务涉及对输入用某种方法进行分析，所以句法分析是计算机程序设计中普遍的组成部分。本章描述单子式的句法分析，主要设计各种表达式的简单句法分析器。本章也介绍一点将分析结果编码为串的逆过程，换言之，介绍关于 Show 的更多内容。这些内容将在最后一章用到。

11.1 单子句法分析器

句法分析器根据需要返回不同的值，所以，可以先把句法分析器看作一个函数输入是串，结果是某种类型的值：

```
type Parser a = String -> a
```

这个类型基本上是标准引导库函数 read 的类型：

```
read :: Read a => String -> a
```

确实，read 就是一个句法分析器，只是它不够灵活。一个原因是它的所有输入必须全部消耗掉。因此，有


```
ghci> read "123" :: Int
123
ghci> read "123+51" :: Int
*** Exception: Prelude.read: no parse
```

使用 read 没有明显的方法完成相继读两个或者更多的输入。例如，可能需要算术表达式的分析器在输入中先找出一个数值，然后是一个运算符，接着是另一个数值。分析数值的第一个分析器将消耗输入的某个前缀，分析运算符的第二个分析器消耗剩余输入的一个前缀，第三个分析器再消耗更多的输入。一个更好的想法是将分析器看作一个函数，该函数消耗输入的一个前缀，并返回一个期望的值和未消耗的输入：

```
type Parser a = String -> (a,String)
```

这个类型还不够周到。一个分析器可能在某些输入上失败。构造可能失败的分析器不是一个错误。例如，对于算术表达式的分析器，我们可能想找一个数值或者一个左括号。一个分析器或者辅助分析器中的一个将会失败。失败不应该看成一个错误，并终止分析进程，而是应该看成一个在两种可能性间的选择运算的单位元。更一般地，一个分析器可能找到对输入的某个前缀的不同分析结果。失败因此对应于分析结果是空序列的情况。为了处理这些不同的可能性，我们将定义再次改为：

```
type Parser a = String -> [(a,String)]
```

标准引导库恰好提供了这个类型同义词，只是在那里这个类型称为 ReadS，不是 Parser。而且，引导库提供了一个函数：

```
reads :: Read a => ReadS a
```

作为类族 Read 的辅助方法。例如：

```
ghci> reads "-123+51" :: [(Int,String)]
[(-123,"+51")]
ghci> reads "+51" :: [(Int,String)]
[]
```

同函数 read 一样，使用 reads 时需要说明期望的类型。上面的第二个例子失败，返回空的分析结果，因为一个 Haskell 整数前面可以冠以一个负号，但是不可以冠以加号。根据定义，如果一个分析器对于所有的可能情况返回空或者单个结果的列表，则称为*确定的*（deterministic）分析器。特别是，reads 的特例应该是确定的分析器。 277

必须对 Parser 的定义做进一步的修改。我们想把这个类型设置成类族 Monad 的实例，但是目前不可行。原因是 Parser 声明为类型同义词，而类型同义词不可以设置成任何类族的成员，它们只能继承该类型所声明的任何实例。类型同义词只是为了改进类型声明的可读性，这里没有任何新类型的创建，因而不能为本质上一样的类型构造两个不同的类族实例。

一种构造新类型的方法是使用数据声明：

```
data Parser a = Parser (String -> [(a,String)])
```

右边的标识符 Parser 是一个构造函数，左边是新类型的名。多数人喜欢这种双关语，也有一些人会用其他标识符命名构造函数，如 MkParser 或者只用 P。

为 Parser 构造一个新类型的更好方法是利用 newtype 声明：

```
newtype Parser a = Parser (String -> [(a,String)])
```

到目前为止，还没有遇到需要 newtype 的情况，所以现在先对此做些解释。利用 data 声明 Parser 类型的代价是查看分析器的运行必须经常用构造函数 Parser 打包和拆包，由此增加分析器的运行时间。另外，还存在一个无用的 Parser 元素，即 Parser undefined。换言之，Parser a 与 String -> [(a,Sring)]是*不同构*（isomorphic）的类型。认识到这一点后，Haskell 允许用 newtype 声明用单个构造函数和单个参数构造的类型。它与类型同义词类型的不同在于，newtype 构造了一个真正的新类型，其元素必须用 Parser 打包。这些强制转换虽然必须出现在程序中，但是并没有增加程序的运行时间，因为编译器在求值前先消去了包装。新类型的值被系统地用底层的类型值代替。因此，Parser a 与 String -> [(a,String)]描述了同构的类型，而且 Parser undefined 与 undefined 是使用同一个表示的同构值。不同于同义词类型的新类型可以说明为类族的成员，而且定义方法可不同于底层的类型。

无论使用哪种类型声明，都必须提供应用分析函数的方法，故定义：

```
apply :: Parser a -> String -> [(a,String)]
apply (Parser p) s = p s
```

278

函数 apply 和函数 Parser 是互逆的，而且是同构映射。

也可定义：

```
parse :: Parser a -> String -> a
parse p = fst . head . apply p
```

函数 parse p 返回第一次分析的第一个结果，如果分析器 p 失败，则引起错误。这也是唯一可能发生错误的地方。

现在可以定义：

```
instance Monad Parser where
  return x = Parser (\s -> [(x,s)])
  p >>= q  = Parser (\s -> [(y,s'')
                           | (x,s') <- apply p s,
                             (y,s'') <- apply (q x) s'])
```

在 p >>= q 的定义中，首先分析器 p 被应用于输入串，生成可能的分析结果与对应未消耗输入二元组列表；然后分析器 q 被应用于 p 的每个分析结果，生成可能结果列表，并将这些列表串联在一起形成最后结果。可以证明，3 个单子律成立。证明留作练习。

11.2 基本分析器

或许最简单的分析器是

```
getc :: Parser Char
getc = Parser f
       where f []     = []
             f (c:cs) = [(c,cs)]
```

如果输入不空，这个分析器返回输入的第一个字符。它的作用恰如第 10 章输入 - 输出单子的 getChar。

下一个分析器识别满足给定条件的一个字符：

```
sat :: (Char -> Bool) -> Parser Char
sat p = do {c <- getc;
            if p c then return c
279         else fail}
```

其中 fail 定义如下：

```
fail = Parser (\s -> [])
```

分析器 fail 是另一个不返回结果的基本分析器。分析器 sat p 读一个字符，如果该字符满足条件 p，则返回该字符作为结果。sat 的定义可以利用一个称为 guard 的小组合器写得更简洁：

```
sat p = do {c <- getc; guard (p c); return c}

guard :: Bool -> Parser ()
guard True  = return ()
guard False = fail
```

要检验这两定义是一致的，注意如果 p c 不真，则有

```
guard (p c) >> return c = fail >> return c = fail
```

注意定律 fail >> p = fail 的应用，其证明留作练习。如果 p c 为真，则有

```
 guard (p c) >> return c
= return () >> return c
= return c
```

利用 sat 可以定义其他分析器，例如：

```
char :: Char -> Parser ()
char x = do {c <- sat (==x); return ()}

string :: String -> Parser ()
string []     = return ()
string (x:xs) = do {char x; string xs; return ()}

lower :: Parser Char
lower = sat isLower

digit :: Parser Int
digit = do {d <- sat isDigit; return (cvt d)}
        where cvt d = fromEnum d - fromEnum '0'
```

分析器 char x 在输入串中查找特定的字符 x，分析器 string xs 查找特定的串，如果成功，两个分析器均返回 ()。例如： 280

```
ghci> apply (string "hell") "hello"
[((),"o")]
```

分析器 digit 查找一个数字字符，并在成功的情况下返回对应的整数。分析器 lower 查找一个小写字母，如果成功，则返回这样的字符。

11.3 选择与重复

为了定义更复杂的分析器，需要在多种可能间选择分析器的运算和重复一个分析器的运算。一种这样的选择运算（ <|> ）定义为

```
(<|>) :: Parser a -> Parser a -> Parser a
p <|> q = Parser f
          where f s = let ps = apply p s in
                      if null ps then apply q s
                      else ps
```

因此 p <|> q 返回 p 的分析结果，除非 p 失败。对于 p 失败的情况，则返回 q 的分析结果。如果 p 和 q 都是确定的，那么 p <|> q 也是确定的。对于 <|> 的其他选择，参见习题。可以断言， <|> 满足结合律，单位元为 fail，其证明也放在习题中。

下面是识别一串小写字母的分析器：

```
lowers :: Parser String
lowers = do {c <- lower; cs <- lowers; return (c:cs)}
         <|> return ""
```

要理解这个分析器如何工作，假定输入是串“Upper”。此时 <|> 左边的分析器失败，因为“U”不是小写字母。但是，右边的分析器成功，所以有

```
ghci> apply lowers "Upper"
[("","Upper")]
```

对于输入“isUpper”，左边分析器成功，故有

```
ghci> apply lowers "isUpper"
[("is","Upper")]
```

281

选择操作符 <|> 的使用需要谨慎。例如，考虑一个非常简单的算术表达式，表达式或者是一位数字，或者是一个数字后接一个加号，再接另一个数字。下面是一个可能的分析器：

```
wrong :: Parser Int
wrong = digit <|> addition

addition :: Parser Int
addition = do {m <- digit; char '+'; n <- digit;
               return (m+n)}
```

有下列结果：

```
ghci> apply wrong "1+2"
[(1,"+2")]
```

分析器 digit 成功，故 addition 没有执行。但是，我们真正想要的是返回 [(3,"")]，以得到尽可能多的输入。纠正 wrong 的一种方法是重写定义如下：

```
better = addition <|> digit
```

此时对于输入 1+2，分析器 addition 成功，返回了我们想要的结果。better 存在的问题是它的效率低：将分析器应用于输入 1 时分析数字成功，但是如果没有找到后续的加号，所以分析器 addition 失败。结果是 digit 被激活，输入又一次被从头开始分析。这对一位数字不是问题，但是如果要对一个可能包含很多位的数值进行分析，那么重复的工作量会很大。

最好的解决方法是将两个分析器分量中的数字分析器提取出来：

```
best   = digit >>= rest
rest m = do {char '+'; n <- digit; return (m+n)}
         <|> return m
```

rest 的参数仅仅是一个累积参数。这个方法基本上是第 8 章介绍的方法。提取分析器得到一个公共前缀是改进效率的好主意。

推广 lowers 的定义，可以定义一个分析器组合器，用于重复一个分析器 0 次或者多次：

```
many :: Parser a -> Parser [a]
many p = do {x <- p; xs <- many p; return (x:xs)}
         <|> none
none = return []
```

282

值 none 不同于 fail（为什么?）。现在可以定义：

```
lowers = many lower
```

在许多应用中，所谓的空白（white space）（空格序列、换行符和制表符）可以出现在标记（tokens）（标识符、数值、左括号和右括号等）之间以增强文本的可读性。分析器 space 用于识别空白：

```
space :: Parser ()
space = many (sat isSpace) >> return ()
```

函数 isSpace 定义在 Data.Char 中。下列函数在识别一个给定串前忽略空白：

```
symbol :: String -> Parser ()
symbol xs = space >> string xs
```

更一般地，可以定义在激活一个分析器前忽略空白：

```
token :: Parser a -> Parser a
token p =  space >> p
```

注意到

```
token p <|> token q = token (p <|> q)
```

但是右边的分析器更高效，因为如果第一个分析器失败时，它不会重复查找空白。

有时需要重复一个分析器一次或者多次，而不是 0 次或者多次。这个功能可以利用称为 some（有的分析库称之为 many1）的组合器实现：

```
some :: Parser a -> Parser [a]
some p = do {x <- p; xs <- many p; return (x:xs)}
```

这个定义重复了 many 定义中选择运算的第一个分析器，利用这个事实也可以重新用 some 定义 many：

```
many :: Parser a -> Parser [a]
many p = optional (some p)

optional :: Parser [a] -> Parser [a]
optional p = p <|> none
```

283

现在分析器 some 和 many 是相互递归的。

下面是一个自然数分析器，允许数字前面有空白：

```
natural :: Parser Int
natural = token nat
nat = do {ds <- some digit;
          return (foldl1 shiftl ds)}
      where shiftl m n = 10*m+n
```

辅助的分析器 nat 不允许数字前面有空白。

现在考虑如何定义一个整数分析器，根据定义整数是一个非空数字串，前面可能有负号。读者可能设想下面的分析器可行：

```
int :: Parser Int
int = do {symbol "-"; n <- natural; return (-n)}
      <|> natural
```

但是其效率低（见习题 H），而且或许结果不是我们期望的。例如：

```
ghci> apply int "  -34"
[(-34,"")]
ghci> apply int "  - 34"
[(-34,"")]
```

数字前的空白没有任何问题，但是我们不希望在一个数字和它的负号中间有空白。如

果是这样，以上分析器便有问题。容易修改 int 的定义以给出我们希望的定义：

```
int :: Parser Int
int = do {symbol "-"; n <- nat; return (-n)}
     <|> natural
```

这个分析器仍然是低效的，更好的方法是定义：

```
int :: Parser Int
int = do {space; f <- minus; n <- nat; return (f n)}
 where
 minus = (char '-' >> return negate) <|> return id
```

分析器 minus 返回一个函数，如果第一个符号是负号，则该函数是 negate，否则是恒
284 等函数。

下面再识别中间用逗号分隔，两边用方括号围起来的一个整数列表。假设每个逗号和括号前后都允许有空白，当然一个整数的数字中间不可有空白。以下是一个简短定义：

```
ints :: Parser [Int]
ints = bracket (manywith (symbol ",") int)
```

辅助分析器 bracket 用来处理括号：

```
bracket :: Parser a -> Parser a
bracket p = do {symbol "[";
                x <- p;
                symbol "]";
                return x}
```

函数 manywith sep p 有些像 many p，不同之处在于 p 的实例是由 sep 的实例分隔的，后者的结果被忽略。其定义如下：

```
manywith :: Parser b -> Parser a -> Parser [a]
manywith q p = optional (somewith q p)

somewith :: Parser b -> Parser a -> Parser [a]
somewith q p = do {x <- p;
                   xs <- many (q >> p);
                   return (x:xs)}
```

例如：

```
ghci> apply ints "[2, -3, 4]"
[([2,-3,4],"")]
ghci> apply ints "[2, -3, +4]"
[]
ghci> apply ints "[]"
[([],"")]
```

整数前面不能冠以加号，所以第二个表达式的解析失败。

11.4 语法与表达式

285 到目前为止，所介绍的组合运算足以将所要求的结构描述翻译成函数分析器。这种结构描述用一个*语法*（grammar）提供。下面将通过各种算术表达式的分析器来展示一些典型的语法。

首先构建下面定义的类型 Expr 的分析器：

```
data Expr = Con Int | Bin Op Expr Expr
data Op   = Plus | Minus
```

下面是一个完全使用括号的语法，用所谓的*巴科斯－诺尔范式*（Backus-Naur form），简称 BNF：

```
expr  ::= nat | '(' expr op expr ')'
op    ::= '+' | '-'
nat   ::= {digit}+
digit ::= '0' | '1' | ... | '9'
```

这个语法定义了 4 个*语法范畴*（syntactic categories）。引号中的符号称为*终结*（terminal）符，而且符号本身是自描述的。这些符号是实际出现在文本中的符号。数字符号有 10 种可能，一个 nat 定义为一个或者多个数字的序列。元符号{ - } + 描述一个语法范畴的非零次重复。注意，这里不允许在数字序列前出现可选的负号，所以，常数是自然数，不是任意整数。语法表明，一个表达式或者是一个自然数，或者是一个复合表达式，其构成是左括号后接一个表达式，然后是一个加号或者减号，接着是另一个表达式，最后是右括号。对描述的隐含理解为终结符之间的空白可以忽略，但是一个数的数字间的空格不能忽略。这个语法可以直接翻译成表达式的分析器：

```
expr :: Parser Expr
expr = token (constant <|> paren binary)
constant = do {n <- nat; return (Con n)}
binary = do {e1 <- expr;
             p <- op;
             e2 <- expr;
             return (Bin p e1 e2)}
op = (symbol "+" >> return Plus) <|>
     (symbol "-" >> return Minus)
```

为了增加可读性，引入辅助分析器 binary。分析器 paren 的定义留作练习。

假定需要一个能够识别省略括号的表达式，如 6 - 2 - 3、6 - (2 - 3) 和 (6 - 2) - 3 等。此时，表达式中的 (+) 和 (-) 应该左结合，就像普通的算术表达式一样。用 BNF 表达这种语法的一种方法如下： 286

```
expr ::= expr op term | term
term ::= nat | '(' expr ')'
```

这个语法表示，一个表达式是一个项或者项之间用运算符分隔的多个项的序列。一个项或者是一个数，或者是一个用括号包围的表达式。特别是，6 - 2 - 3 将被分析成表达式 6 - 2 后接一个减号，然后再接项 3。换句话说，结果等同于 (6 - 2) - 3，这也是我们要求的。这个语法也可以直接翻译成一个分析器：

```
expr = token (binary <|> term)
binary = do {e1 <- expr;
             p  <- op;
             e2 <- term;
             return (Bin p e1 e2)}
term = token (constant <|> paren expr)
```

但是，这个分析器有一个致命问题：它会陷入无穷循环。分析器 expr 的第一个动作首先忽略空白，然后调用 binary，而 binary 的第一个动作又是调用 expr。糟糕！

另外，如下定义 expr 是不可行的。

```
expr = token (term <|> binary)
```

因为，例如：

```
Main*> apply expr "3+4"
[(Con 3,"+4")]
```

只有第一个项被识别。这种问题称为*左递归*（left recursion）问题，而且是所有递归分析器会遇到的问题，包括函数式的以及其他的分析器。

一个解决方法是用下列等价形式重写语法：

```
expr ::= term {op term}*
```

其中的元符号{ - }* 表示一个语法范畴被重复 0 次或者多次。此时新的分析器具有下列形式：

```
expr = token (term >>= rest)
rest e1 = do {p <- op;
              e2 <- term;
              rest (Bin p e1 e2)} <|> return e1
```

287

分析器 rest 对应于范畴{op term}* ，而且它有一个参数（累积参数），其值是当前分析的结果。

最后，设计一个算术表达式的分析器，表达式可以包含乘法和除法，并如下修改 Op 的定义：

```
data Op = Plus | Minus | Mul | Div
```

采用普通的运算规则，乘法和除法优先级高于加法和减法的优先级，相同优先级的运算向左结合。下面是语法：

```
expr ::= term {addop term}*
term ::= factor {mulop factor}*
factor ::= nat | '(' expr ')'
addop ::= '+' | '-'
mulop ::= '*' | '/'
```

其分析器如下：

```
expr = token (term >>= rest)
rest e1 = do {p <- addop;
              e2 <- term;
              rest (Bin p e1 e2)}
          <|> return e1
term = token (factor >>= more)
more e1 = do {p <- mulop;
              e2 <- factor;
              more (Bin p e1 e2)}
          <|> return e1
factor = token (constant <|> paren expr)
```

分析器 addop 和 mulop 的定义留作习题 K。

11.5 显示表达式

最后一个问题是，如何将 Expr 设置成类族 Show 的成员，使得函数 show 成为语法

分析的逆函数？更确切地说，定义 show 使得

```
parse expr (show e) = e
```

注意，parse p 抽取由 apply p 返回的第一个分析结果。 288

作为热身准备，下面是 Expr 作为 Show 的实例定义，假定 expr 是只包含加减运算及完全括号表达式的分析器：

```
instance Show Expr where
  show (Con n) = show n
  show (Bin op e1 e2) =
    = "(" ++ show e1 ++
      " " ++ showop op ++
      " " ++ show e2 ++ ")"
showop Plus = "+"
showop Minus = "-"
```

定义很清楚，但是效率有问题。因为（++）的运行时间是其左边参数的线性函数，所以，在最坏情况下 show 的求值运行时间是表达式规模的二次方函数。

解决方法仍然是使用累积参数。Haskell 提供了一个类型别名 ShowS：

```
type ShowS = String -> String
```

以及下列辅助函数：

```
showChar   :: Char -> ShowS
showString :: String -> ShowS
showParen  :: Bool -> ShowS -> ShowS
```

这些函数定义如下：

```
showChar      = (:)
showString    = (++)
showParen b p = if b then
                  showChar '(' . p . showChar ')'
                  else p
```

现在可以给出表达式的 show 函数如下：

```
show e = shows e ""
 where
 shows (Con n) = showString (show n)
 shows (Bin op e1 e2)
  = showParen True (shows e1 . showSpace .
    showsop op . showSpace  . shows e2)
showsop Plus  = showChar '+'
showsop Minus = showChar '-'
showSpace     = showChar ' '
```

289

这个版本没有显式使用串联运算，运行时间是表达式规模的线性函数。

假如现在要显示省略括号的表达式。左边表达式的括号可以省略，但是右边表达式的括号则不能省略。由此得到

```
show = shows False e ""
 where
 shows b (Con n) = showString (show n)
 shows b (Bin op e1 e2)
  = showParen b (shows False e1 . showSpace .
    showsop op . showSpace  . shows True e2)
```

这个定义没有考虑到结合性，例如，1 +(2 +3) 的显示结果不是1 +2 +3。

最后，考虑包含所有4种运算的表达式。这里的区别在于：

1. 对于表达式 e1 + e2 或者 e1 - e2，无需在 e1 周围加括号（同上面情况一样），如果 e2 是一个以乘法或除法为根运算的复合表达式，那么 e2 周围也不需要加括号。

2. 另一方面，对于表达式 e1 *e2 或者 e1 / e2，如果 e1 是根运算为加法或者减法的表达式，需要在 e1 周围加括号，而且 e2 周围总是需要加括号。

对这些规则编码的一种方法是引入优先级（另一种方法见习题L）。定义：

```
prec :: Op -> Int
prec Mul   = 2
prec Div   = 2
prec Plus  = 1
prec Minus = 1
```

现在考虑如何定义下列类型的函数 showsPrec：

```
showsPrec :: Int -> Expr -> ShowS
```

使得 showsPrec p e 能够显示表达式 e，假定 e 的父结点是优先级为 p 的运算的复合表
290 达式。然后函数 show 可以定义为

```
show e = showsPrec 0 e ""
```

使得围绕 e 的上下文（context）是一个假想优先级0的运算。马上可以定义：

```
showsPrec p (Con n) = showString (show n)
```

因为常数永远不需要括号。有趣的是复合表达式的情况，这里先给出定义，然后再解释：

```
showsPrec p (Bin op e1 e2)
 = showParen (p>q) (showsPrec q e1 . showSpace .
   showsop op . showSpace . showsPrec (q+1) e2)
   where q = prec op
```

如果父运算符的优先级高于当前运算优先级，则在一个表达式周围加括号。要显示表达式 e1，只需将当前优先级作为新的父优先级传递下去。但是，如果 e2 的根运算优先级低于或者等于 q，则在 e2 周围需要加括号，所以在第二次调用中对 q 加1。

不可否认的是，showsPrec 的以上定义需要一些思考，但这也是值得的。类族 Show 有第二个方法，即 showsPrec。而且，show 的缺省定义正是如上的定义。所以，将表达式作为 Show 的成员，只需给出 showsPrec 的定义即可。

11.6 习题

习题A 考虑类型同义词：

```
type Angle = Float
```

假设在 Angle 上定义的相等为模 2π 的某个倍数相等。请问为什么不能用（==）进行这样的测试？再考虑：

```
newtype Angle = Angle Float
```

291 请定义 Angle 为 Eq 的成员，使得（==）可用来判断两个 Angle 元素是否相等。

习题 B 可以定义：

```
newtype Parser a = Parser (String -> Maybe (a,String))
```

请给出这种分析器的单子实例定义。

习题 C 证明 `fail >> p = fail`。

习题 D 可否如下定义 `<|>`？

```
p <|> q = Parser (\s -> parse p s ++ parse q s)
```

在什么情况下结果是一个确定的分析器？请定义一个函数：

```
limit :: Parser a -> Parser a
```

使得即使 `p` 和 `q` 不是确定的，`limit (p <|> q)` 也是一个确定的分析器。

习题 E 分析器不仅是单子的实例，也可以是之前介绍的、更局限的称为 `MonadPlus` 类族的实例。它们基本上是支持选择和失败的单子。Haskell 定义为

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

例如，列表类型构造函数 `[]` 和 `Maybe` 都可以定义为 `MonadPlus` 的实例：

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` y = y
  Just x  `mplus` y = Just x
```

请将 `Parser` 定义为 `MonadPlus` 的实例。 292

习题 F 接着习题 E，新的方法 `mzero` 和 `mplus` 应该满足一些等式定律，就像通常一个类族的方法一样。但是，目前 Haskell 设计者对于这些方法应该满足的确切定律没有一致意见。没有争议的定律是 `mplus` 具有单位元 `mzero`，并且满足结合律。由此得到 3 条定律。另一条合理的定律是*左零元*（left-zero）律：

```
mzero >>= f = mzero
```

相应的*右零元*（right-zero）律也应该满足：

```
p >> mzero = mzero
```

请问列表单子作为 `MonadPlus` 的实例满足这 5 条定律吗？

最后，真正有争议的定律是下面的等式：

```
(p `mplus` q) >>= f = (p >>= f) `mplus` (q >>= f)
```

这条定律称为*左分配*（left-distribution）律。如果要求满足左分配律，那么 `Maybe` 为什么不能成为 `MonadPlus` 的成员？

习题 G 设计一个识别 Haskell 浮点数的分析器。记住，`.314` 不是合法的数字（小数点前没有数字），而且 `3. 4` 也是不合法的（小数点前后不允许空白）。

习题 H 对比 `int` 的 3 个定义，为什么本书中 `int` 的第一个定义和第二个定义是低

效的?

习题 I 请问"(3)"是括号完全的表达式吗?它是省略括号的表达式吗?Haskell 允许常数带括号:

```
ghci> (3)+4
7
```

293 请设计一个括号完全的表达式分析器,而且允许在常数周围加括号。

习题 J 考虑语法 `expr ::= term {op term}*` 。请定义 `pair` 和 `shunt` 使得下列分析器是合理的:

```
expr = do {e1 <- term;
           pes <- many (pair op term);
           return (foldl shunt e1 pes)}
```

习题 K 请给出 `addop` 和 `mulop` 的定义。

习题 L 再考虑包含 4 种算术运算的表达式的显示问题。使用括号的规则:在表达式 `e1 op e2` 中,如果 `op` 是乘法运算符,而且 `e1` 的根不是乘法运算符,则在 `e1` 周围需要括号。对偶地,如果 `op` 是乘法运算符,或者 `e2` 的根不是乘法运算符,那么 `e2` 周围需要括号。定义:

```
isMulOp Mul = True
isMulOp Div = True
isMulOp _   = False
```

请构造 `show` 的另一个定义,并使用下列辅助函数:

```
showsF :: (Op -> Bool) -> Expr -> ShowS
```

11.7 答案

习题 A 答案 因为 `(==)` 是浮点数上的相等测试,不同的浮点数不相等。

```
instance Eq Angle where
 Angle x == Angle y = reduce x == reduce y
  where
  reduce x | x<0 = reduce (x + r)
           | x>r = reduce (x - r)
           | otherwise = x
294        where r = 2*pi
```

习题 B 答案

```
instance Monad Parser where
  return x = Parser (\s -> Just (x,s))
  P >>= q  = Parser (\s -> case apply p s of
                             Nothing -> apply q s
                             Just (x,s') -> Just (x,s'))
```

习题 C 答案

```
  fail >> p
= fail >>= const p
= fail
```

根据 `fail` 的定义和 `p >>= q` 的定义立即可以得出 `fail >>= p = fail` 的事实。

习题 D 答案 是的，可以，但是只有当 p 或者 q 是 fail 时，结果才是确定的。函数 limit 可以如下定义：

```
limit p = Parser (take 1 . apply p)
```

习题 E 答案

```
mzero = fail
mplus = (<|>)
```

习题 F 答案 是的，列表单子和 Maybe 单子均满足这 5 条定律。例如，对于列表单子：

```
mzero >>= f = concat (map f []) = [] = mzero
xs >> mzero = concat (map (const []) xs) = [] = mzero
```

对于 Maybe，左分配律不成立。因为

```
  (Just x `mplus` q) >>= (\x -> Nothing)
= Just x >>= (\x -> Nothing)
= Nothing
```

但是 295

```
  (Just x >> \x -> Nothing) `mplus`
  (q >>= \x -> Nothing)
= Nothing `mplus` (q >>= \x -> Nothing)
= q >>= \x -> Nothing
```

两个结果表达式不相等（take q = undefined）。

习题 G 答案

```
float :: Parser Float
float = do {ds <- some digit;
            char '.';
            fs <- some digit;
            return (foldl shiftl 0 ds +
                    foldr shiftr 0 fs)}
        where shiftl n d = 10*n + fromIntegral d
              shiftr f x = (fromIntegral f+x)/10
```

分析器 digit 返回一个 Int，这个 Int 必须转换成一个数（在这里是 Float）。

习题 H 答案 空白被分析了两次。例如，用 int1 和 int3 分别表示第一个和第三个版本，则有

```
ghci> apply int3 $ replicate 100000 ' ' ++ "3"
[(3,"")]
(1.40 secs, 216871916 bytes)
ghci> apply int1 $ replicate 100000 ' ' ++ "3"
[(3,"")]
(2.68 secs, 427751932 bytes)
```

习题 I 答案 不是，根据 expr 的第一个语法，只有二元表达式可以用括号。是的，根据第二个定义，任何表达式都可以用括号围起来。

修改后的语法是

```
expr ::= term | '(' expr op expr ')'
term ::= nat | '(' expr ')'
```

296 相应的分析器是

```
expr = token (term <|> paren binary)
 where
 term = token (constant <|> paren expr)
 binary = do {e1 <- expr;
              p <- op;
              e2 <- expr;
              return (Bin p e1 e2)}
```

习题 J 答案

```
pair :: Parser a -> Parser b -> Parser (a,b)
pair p q = do {x <- p; y <- q; return (x,y)}

shunt e1 (p,e2) = Bin p e1 e2
```

习题 K 答案

```
addop = (symbol "+" >> return Plus) <|>
        (symbol "-" >> return Minus)
mulop = (symbol "*" >> return Mul) <|>
        (symbol "/" >> return Div)
```

习题 L 答案

```
show e = showsF (const False) e ""
 where
 showsF f (Con n) = showString (show n)
 showsF f (Bin op e1 e2)
  = showParen (f op) (showsF f1 e1 . showSpace .
    showsop op . showSpace . showsF f2 e2)
    where f1 x = isMulOp op && not (isMulOp x)
          f2 x = isMulOp op || not (isMulOp x)
```

11.8 注记

用单子方式设计函数式语法分析器一直以来都是很受欢迎的函数式程序设计的应用。本章的内容沿用 Graham Hutton 和 Erik Meijer 的“Monadic parsing in Haskell”，参见 the
297 Journal of Functional Programming8(4)，437-144，1998。

一个简单的等式计算器

最后一章介绍一个程序设计项目，设计并实现一个执行点自由等式证明的简单计算器。尽管计算器只提供一个自动证明辅助器中部分的功能，并在许多方面很局限，但是，这个计算器足以证明之前描述的许多点自由定律，当然，只要我们准备好在必要的时候在正确的方向上助其一臂之力。本项目也是使用模块系统的实例。计算器的每个组成部分，相关类型和函数，都定义在一个适当的模块中，并通过显式的输入和输出关联模块。

12.1 基本思想

基本想法是设想具有下列类型的函数 `calculate`：

```
calculate :: [Law] -> Expr -> Calculation
```

函数 `calculate` 的第一个参数是可能应用的定律列表。每个定律由一个描述性的名称和一个方程组成。第二个参数是一个表达式，结果是一个计算。一个计算由一个开始表达式和一系列步骤构成。每一步由一个定律的名和将该定律左边应用于当前表达式得到的表达式构成。当没有定律可用时，计算结束，最后的表达式便是计算的结果。整个过程是自动的，无需用户参与。

298

定律、表达式和计算都是将在下面几节定义的适当数据类型的元素。不过，现在先插入一个例子，说明心目中的框架。

下面是几个定律（使用小字体避免断行）：

```
definition filter:  filter p = concat . map (box p)
definition box:     box p = if p one nil

if after dot:       if p f g . h = if (p . h) (f . h) (g . h)
dot after if:       h . if p f g = if p (h . f) (h . g)

nil constant:       nil . f = nil
map after nil:      map f . nil = nil
map after one:      map f . one = one . f

map after concat:   map f . concat = concat . map (map f)

map functor:        map f . map g = map (f . g)
map functor:        map id = id
```

每个定律由一个名和一个等式构成。定律的名用冒号表示结束，一个等式由两个表达式中间用等号分隔表示。每个表达式描述一个函数，计算器只用于简化这些函数表达式（是的，是点自由计算器）。表达式由常数（如 `one` 和 `map`）和变量（如 `f` 和 `g`）构成。确切的语法将在适当的时候说明。注意，没有条件定律，即只有在某些辅助条件满足的条件下成立的等式。这将限制计算器所能做的工作，但是，这个计算器仍然是非常有趣的。

假如要简化表达式 `filter p . map f`。以下是一种可能的计算：

```
  filter p . map f
=   {definition filter}
  concat . map (box p) . map f
=   {map functor}
  concat . map (box p . f)
=   {definition box}
  concat . map (if p one nil . f)
=   {if after dot}
  concat . map (if (p . f) (one . f) (nil . f))
=   {nil constant}
  concat . map (if (p . f) (one . f) nil)
```

299 计算步骤用常规方式显示，所使用的定律写在括号中，并放在它应用的表达式之间。没有定律可以应用于最后的表达式，该表达式就是计算的结果。显然，结果不比开始要化简的表达式简单。

计算器有可能以不同的顺序应用某些定律。例如，box 的定义可以在第二步使用，而不是在第三步使用。但是，最后的结果是相同的。也有可能，尽管不是这组定律，一个表达式用不同的计算得到不同的结果。但是，从一开始我们就决定 calculate 返回一个结果，而不是所有可能计算组成的树。

注意每一步计算在发生什么。某个定律的左边与当前表达式的某个子表达式进行匹配。如果匹配成功，那么结果是定律中变量的代换（substitution）。例如，在第二步中，子表达式 map (box p) . map f 与第一个 map 函子律成功匹配，结果是一个代换，其中 map 函子律中的变量 f 绑定到 box p，变量 g 绑定到 f。这一步的结果是重写子表达式，首先将定律右式中的变量用其绑定的表达式代换，然后用代换后的右式代替子表达式。匹配、代换和重写都是计算器的基本组成部分。

现在假定使用以上同一组定律化简表达式 map f . filter (p . f)。下面是计算过程：

```
  map f . filter (p . f)
=   {definition filter}
  map f . concat . map (box (p . f))
=   {map after concat}
  concat . map (map f) . map (box (p . f))
=   {map functor}
  concat . map (map f . box (p . f))
=   {definition box}
  concat . map (map f . if (p . f) one nil)
=   {dot after if}
  concat . map (if (p . f) (map f . one) (map f . nil))
=   {map after nil}
  concat . map (if (p . f) (map f . one) nil)
=   {map after one}
  concat . map (if (p . f) (one . f) nil)
```

同样，某些定律有可能按照不同的次序应用。对最后的表达式没有可用的定律，所以成为计算的结果。

300 这里要说明的重点是，两个计算的最后表达式是一样的，所以证明了

```
filter p . map f = map f . filter (p . f)
```

这就是将要进行的等式证明的方法，将两边简化为同一个结论。代之以将两个计算一个接一个显示出来，另一种方法是将两个计算结果粘在一起，先记录第一个计算过程，然后附上第二个计算的逆过程。这种方法的主要优点是简单，为了达到期望的目标，不必发明一种新的证明形式，也不一定要按照从右到左的方式使用定律。所以，我们还将定义一个证

明等式的函数：

```
prove :: [Law] -> Equation -> Calculation
```

进一步的考虑

计算器的一个基本约束是定律只能从一个方向应用，即从左至右应用定律。这个约束主要是为了避免计算陷入循环。如果允许在两个方向应用定律，那么计算器可能在一个方向应用一个定律，然后立即从反方向应用该定律，从而来回摆动。

即使规定从左到右的规则，有些定律仍然可能导致无限的计算。典型的情况是递归函数定义的定律。例如，考虑 iterate 的定义：

```
defn iterate: iterate f = cons . fork id (iterate f . f)
```

这是点自由形式的 iterate 定义。函数 cons 和 fork 定义如下：

```
cons (x,xs) = x:xs
fork f g x  = (f x,g x)
```

在第 4 章和第 6 章的习题中已经遇到过 fork，只是那里写成 fork (f,g) 而不是这里的 fork f g。以下的定义都采用卡瑞式。项 iterate f 在定律两边出现的形式表示，如果一个计算可以使用 iterate 的定义一次，那也意味着这个定义可以潜在地被使用无限次。但是，这种事也不是必然的。下面的计算（计算器生成的）就避免了无穷次的回归：

```
  head . iterate f
=   {defn iterate}
  head . cons . fork id (iterate f . f)
=   {head after cons}
  fst . fork id (iterate f . f)
=   {fst after fork}
  id
```

301

这个计算使用了下面两个定律：

```
head after cons:  head . cons = fst
fst after fork:   fst . fork f g = f
```

这个计算能够避免无穷计算的原因是这两个定律有比定义高的优先级，后面将会进一步说明这个技巧。

为了说明计算器的能力，能做什么，不能做什么，下面是另一个将递归定义翻译成点自由形式的例子。考虑串联的定义：

```
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

下面将用 cat 表示 (++)。另外还需要 nil、cons 和函数 cross (f,g)，并用 f *g表示后者。因此，有

```
(f * g) (x,y) = (f x, g y)
```

最后，还需要一个组合子 assocr（associate-right 的简写），其定义为

```
assocr ((x,y),z) = (x,(y,z))
```

下面是 cat 两个定义方程的点自由式翻译：

```
cat . (nil * id)  = snd
cat . (cons * id) = cons . (id * cat) . assocr
```

不能用这里的计算器证明 cat 满足结合律，因为证明涉及归纳证明，但是，可以将其陈述为一个定律：

```
cat associative: cat . (cat * id) = cat . (id * cat) . assocr
```

继续深入这个例子，下面是（*）的两个双函子定律：

```
bifunctor *:     id * id = id
bifunctor *:     (f * g) . (h * k) = (f . h) * (g . k)
```

下面是关于 assocr 的定律：

302

```
assocr law:  assocr . ((f * g) * h) = (f * (g * h)) . assocr
```

对于这个例子，计算器不能完成下列的合法计算：

```
  cat . ((cat . (f * g)) * h)
=   {identity law, in backwards direction}
  cat . ((cat . (f * g)) * (id . h))
=   {bifunctor *, in backwards direction}
  cat . (cat * id) . ((f * g) * h)
=   {cat associative}
  cat . (id * cat) . assocr . ((f * g) * h)
=   {assoc law}
  cat . (id * cat) . (f * (g * h)) . assocr
=   {bifunctor *}
  cat . ((id . f) * (cat . (g * h))) . assocr
=   {identity law}
  cat . (f * (cat . (g * h))) . assocr
```

问题在于恒等律和双函子律必须在两个方向使用，但是计算器不能完成这样的工作。注意到证明的本质在于下列表达式的两种不同简化：

```
cat . (id * cat) . assocr . ((f * g) * h)
```

一种使用 cat 的结合律，其表现形式为

```
cat associative: cat . (id * cat) . assocr = cat . (cat * id)
```

另一种使用 assocr 定律。即使推广 calculate 使其返回所有可能计算构成的树，但从哪个表达式开始可以得到以上计算仍然不是明显的，所以禁止了计算器返回一棵树。

不只是函子律有时需要从两个方向使用，例子见 12.8 节。这个问题有时可以避免，比如使用比实际要求更通用的定律，或者通过分析，然而有时还是完全不能避免。如一开始所讲，计算器能力是有限的。

我们设计的自动计算只有两个自由度：选择应用的定律和选择化简的子表达式。第一个自由度可以包含在这些定律在计算器中排列的顺序中：如果有两个不同定律可应用，那么选择前一个定律。

显然有些定律应该在另一些定律之前应用，先应用的应该是可以降低中间表达式复杂度的定律。很好的例子是定律 f . id = f 和 id . f = f。复杂度的简单定义是右边的复

303

合比左边的少。这些定律一旦有机会就使用不会有错误。事实上，id 是复合的单位元，而且将被写进计算器，这样两个单位元律将被自动应用。类似地，及早应用如 nil . f = nil 和 map f.nil = nil 这样的定律（确实是 iterate 计算中使用的两个定律）将降低复合次数，有助于降低中间表达式的规模。为了明确起见，称这些定律为简单定律。

另一方面，某些定律只是到最后才使用。典型的是定义，如 filter 或者 iterate 的定义。例如，在下列表达式中：

```
map f . concat . map (filter p)
```

不想太早使用 filter 的定义；而且在 concat 定律后先使用 map，只有在后面必要的时候再使用 filter 的定义。不说别的，中间表达式将会更短。

综上所述，看起来合理的方法是将定律按照“简单定律、非简单也非定义的定律、定义”的顺序应用。

第二个自由度用给定表达式中子表达式作为定律应用特例的顺序表达：如果定律可应用于两个不同的表达式，那么先选列在前面的子表达式。

现在仍然没有决定使用定律和子表达式的优先次序。是先选一个子表达式，然后轮流检查每个定律是否可用于该子表达式，还是先选一个定律，然后检查这个定律可用于哪个子表达式？或许将某个定律应用于一个表达式后，下一个利用的定律很可能是“旁边的”某个子表达式，但是如何表达这种附近的概念不是很清楚，从计算时间或者结果的长度上讲，这样做是否会提高计算的效率也不清楚。

12.2 表达式

计算器的核心是表达式的数据类型 Expr。计算器的许多配件都需要通过各种方式分析和处理表达式。表达式是由（函数）变量和常量构成的，函数复合运算是基本的组合形式。变量没有参数，但是常量可以有任意多参数，这些参数本身也是表达式。假定所有函数是卡瑞式的，没有多元组参数，如写 pair f g，而不写 pair (f,g)。没有特别的理由避免使用多元组，只是本书讨论的大多数函数都是卡瑞式的，没有必要两种形式都使用。 304

为了补偿，允许使用中缀二元运算，如写 f *g，而不写 cross f g。除函数复合运算外，对二元运算的优先级或结合性不做任何假设，当表达式包含这些运算时全部使用括号。这里仍然没有回答的问题是，f *g . h 表示(f *g) . h 还是 f *(g . h)？Haskell 规定函数复合具有更高优先级，我们遵循这个规则。所以，f *g . h 被视为 f *(g . h)。但是，我们会永远使用括号以避免歧义。

下面是表达式的 BNF 语法：

```
expr   ::= simple {op simple}
simple ::= term {'.' term}*
term   ::= var | con {arg}* | '(' expr ')'
arg    ::= var | con | '(' expr ')'
var    ::= letter {digit}
con    ::= letter letter {letter | digit}*
op     ::= {symbol}+
```

变量名由单个字母表示，也可以后接一位数字。所以，f 和 f1 都是合法的变量名。常量名由两个以上字母数字序列构成，用两个字母开始，如 map 和 lhs2tex，而运算名用非字母数字序列表示，如* 和 <+>。语法的第一行表示，一个表达式是一个简单表达式，可以后接一个运算符和另一个简单表达式。简单表达式是项的组合。相信剩余的行是容易理解的。

下面是要使用的 Expr 的定义：

```
newtype Expr = Compose [Atom] deriving Eq
data Atom    = Var VarName | Con ConName [Expr]
               deriving Eq
type VarName = String
type ConName = String
```

305

表达式和原子声明为类族 Eq 的成员，因为需要测试表达式是否相等。稍后将定义 Expr 为类族 Show 的实例，以便将表达式打印在终端。

以下是一些表达式及其表示的例子：

```
f . g . h   => Compose [Var "f",Var "g",Var "h"]
id          => Compose []
fst         => Compose [Con "fst" []]
fst . f     => Compose [Con "fst" [],Var "f"]
(f * g) . h => Compose [Con "*" [Compose [Var "f"],Compose [Var "g"]],Var "h"]
f * g . h   => Compose [Con "*" [Compose [Var "f"],
                                 Compose [Var "g",Var "h"]]]
```

函数复合运算满足结合律的性质已经定义在 Expr 中。特殊的常数 id 保留，并总是解释为复合的单位元。

前面描述的语法分析组合运算使我们得以分析表达式。根据 BNF 定义，首先有

```
expr :: Parser Expr
expr = simple >>= rest
 where
 rest s1 = do {op <- operator;
               s2 <- simple;
               return (Compose [Con op [s1,s2]])}
           <|> return s1
```

一个运算符是一个或者多个运算符号的序列，只要不含复合运算符和等号：

```
operator :: Parser String
operator = do {op <- token (some (sat symbolic));
               Parsing.guard (op /= "." && op /= "=");
               return op}

symbolic = (`elem` opsymbols)
opsymbols = "!@#$%&*+./<=>?\\^|:-~"
```

函数 Parsing.guard 是一个受限（qualified）名的例子。Haskell 的引导库 Prelude 也提供了一个名为 guard 的函数，但是这里需要的是包含所有语法分析函数的模块

306 Parsing 的同名函数。一个受限名由一个模块名后接一个句点，再后接受限值的名构成。

一个简单表达式由一个或者多个项的序列构成，项之间用复合运算符分隔：

```
simple :: Parser Expr
simple = do {es <- somewith (symbol ".") term;
             return (Compose (concatMap deCompose es))}
```

函数 concatMap f 是 concat . map f 的替代函数，由标准引导库提供，而 deCompose 如下定义：

```
deCompose :: Expr -> [Atom]
deCompose (Compose as) = as
```

接下来，一个项是表示变量或者常量的标识符，常量可以带参数，或者是加括号的表达式：

```
term :: Parser Expr
term = ident args <|> paren expr
args = many (ident none <|> paren expr)
```

分析器 ident 输入一个表达式列表的分析器，返回表达式的一个分析器：

```
ident :: Parser [Expr] -> Parser Expr
ident args
 = do {x <- token (some (sat isAlphaNum));
       Parsing.guard (isAlpha (head x));
       if isVar x
       then return (Compose [Var x])
       else if (x == "id")
       then return (Compose [])
       else
       do {as <- args;
           return (Compose [Con x as])}}
```

测试是否合理变量的函数如下定义：

```
isVar [x]   = True
isVar [x,d] = isDigit d
isVar _     = False
```

注意，任何完全由字母数字组成，而且由字母开始又不是变量的标识符是常量。

接下来将 Expr 和 Atom 定义成 Show 的实例。如前所述，我们将通过给每个类型定义函数 showsPrec p 来完成。稍加思考便可发现，p 需要 3 个值： 307

- 顶层无需括号。例如，所有这些式子 map f . map g，foo *baz 和 bar bie doll 都不需要括号。将 p =0 赋予这种情况。
- 当一个表达式是项的复合，或者运算符表达式，并且是一个常量的参数时，需要将表达式括起来。例如，在下列表达式中需要括号：

  ```
  map (f . g) . foo f g . (bar * bar)
  ```

  但是，中间项无需括号。将 p =1 赋予这种情况。
- 最后，p =2 表示应该给项的复合加括号，给运算符表达式以及至少有一个参数的卡瑞函数加括号。例如：

```
map (f . g) . foo (foldr f e) g . (bar * bar)
```

下面给出实例定义。先定义：

```
instance Show Expr where
 showsPrec p (Compose [])  = showString "id"
 showsPrec p (Compose [a]) = showsPrec p a
 showsPrec p (Compose as)
  = showParen (p>0) (showSep " . " (showsPrec 1) as)
```

最后一行使用了如下定义的函数 showSep：

```
showSep :: String -> (a -> ShowS) -> [a] -> ShowS
showSep sep f
 = compose . intersperse (showString sep) . map f
```

工具函数 compose 定义为 compose = foldr (.) id。函数 intersperse :: a -> [a] -> [a]可以在 Data.List 中找到，用于将第一个参数插入到第二个参数元素

之间。例如：

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

showsPrec 的后两个子句中右边 showsPrec 的两次出现表示原子上的相应函数：

```
instance Show Atom where
 showsPrec p (Var v)    = showString v
 showsPrec p (Con f []) = showString f
 showsPrec p (Con f [e1,e2])
  | isOp f = showParen (p>0) (showsPrec 1 e1 . showSpace .
               showString f . showSpace . showsPrec 1 e2)
 showsPrec p (Con f es)
  = showParen (p>1) (showString f . showSpace .
       showSep " " (showsPrec 2) es)

isOp f = all symbolic f
```

308

p =2 在最后一个子句中用到，因为我们想在如 foo (bar bie) doll 中加括号。变量和零元常量无需括号。

一个模块结构

最后一步是将这些定义，或许还有另外的定义，放置在一个模块中。这个模块将包含所有与表达式有关的函数。

构建这样的模块还不能马上完成，因为不清楚在其他模块（如处理定律和计算等模块）中，可能需要表达式的另外哪些函数。但是，目前可以如下声明：

```
module Expressions
  (Expr (Compose), Atom (Var,Con),
   VarName, ConName, deCompose, expr)
where
import Parsing
import Data.List (intersperse)
import Utilities (compose)
import Data.Char (isAlphaNum,isAlpha,isDigit)
```

模块 Expressions 必须存储在文件 Expressions.lhs 中，使得 Haskell 可以找到这个模块的位置。该模块输出类型 Expr 和 Atom 以及它们的构造函数，还输出类型同义词 VarName 和 ConName，以及函数 deCompose 和 expr，所有这些都可能在处理定律的模块中用到。稍后可能在输出列表中添加更多的函数。

接下来是输入。我们输入了语法分析函数的模块 Parsing，以及模块 Data.List 和 Data.Char 中的一些函数。我们还将设置一个模块 Utilities，放置通用工具函数。一个好的工具函数例子是上面定义的 compose，这个函数不是特定于表达式的，可能在其他地方用到，所以将其放在工具模块中。

309

12.3 定律

用下列方法定义定律：

```
data Law      = Law LawName Equation
type LawName  = String
type Equation = (Expr,Expr)
```

一个定律由一个描述名和一个等式构成。分析定律的分析器定义为

```
law :: Parser Law
law = do {name <- upto ':';
          eqn <- equation;
          return (Law name eqn)}
```

语法分析函数 upto c 返回直到 c 但不包含 c 的串，然后丢弃 c，如果找到 c 的话。该函数不含在第 11 章的分析器函数中，但是为了不破坏分析器的抽象性，将其置于模块 Parsing 之中。一种定义方法是

```
upto :: Char -> Parser String
upto c
 = Parser (\s ->
           let (xs,ys) = break (==c) s in
           if null ys then []
           else [(xs,tail ys)])
```

分析器 equation 定义如下：

```
equation :: Parser Equation
equation = do {e1 <- expr;
               symbol "=";
               e2 <- expr;
               return (e1,e2)}
```

或许不需要显示定律，不过下面是其定义：

```
instance Show Law where
 showsPrec _ (Law name (e1,e2))
     = showString name .
       showString ": " .
       shows e1 .
       showString " = " .
       shows e2
```

310

优先级数在 showsPrec 的定义中不需要，故使用了不在意模式。回顾 shows 的输入是一个可打印值，在这里是一个表达式，然后返回一个类型 ShowS 的函数，即类型 String -> String 的同义词。

最后对定律排序：

```
sortLaws :: [Law] -> [Law]
sortLaws laws = simple ++ others ++ defns
 where
 (simple,nonsimple) = partition isSimple laws
 (defns,others)     = partition isDefn nonsimple
```

这个定义利用 Data.List 的列表划分函数 partition：

```
partition p xs = (filter p xs, filter (not . p) xs)
```

各种测试函数定义如下：

```
isSimple (Law _ (Compose as1,Compose as2))
         = length as1 > length as2
isDefn (Law _ (Compose [Con f es], _))
         = all isVar es
isDefn _ = False
isVar (Compose [Var _]) = True
isVar _                 = False
```

测试 isVar 也出现在模块 Expressions 中，但是定义不同。不过这不是问题，因为该函数没有从表达式模块输出。

下面是定律模块的声明：

```
module Laws
  (Law (Law), LawName, law, sortLaws,
   Equation, equation)
where
import Expressions
import Parsing
import Data.List (partition)
```

311

完成如何分析和打印表达式及定律后，现在可以定义两个函数，一个是 calculate 的一个版本，但是它不消耗定律和表达式，而是消耗串：

```
simplify :: [String] -> String -> Calculation
simplify strings string
  = let laws = map (parse law) strings
        e = parse expr string
    in calculate laws e
```

类似地，可定义：

```
prove :: [String] -> String -> Calculation
prove strings string
 = let laws = map (parse law) strings
       (e1,e2) = parse equation string
   in paste (calculate laws e1) (calculate laws e2)
```

这两个函数可以放置在模块 Main 中。将 paste 和 calculate 放置在只与计算相关的模块中，这是 12.4 节的内容。

12.4 计算

计算的定义如下：

```
data Calculation = Calc Expr [Step]
type Step        = (LawName,Expr)
```

先从计算器的关键定义开始，即 calculate 的定义：

```
calculate :: [Law] -> Expr -> Calculation
calculate laws e = Calc e (manyStep rws e)
 where rws e = [(name,e')
               | Law name eqn <- sortedlaws,
                 e' <- rewrites eqn e,
                 e' /= e]
       sortedlaws = sortLaws laws
```

函数 rewrite :: Equation -> Expr -> [Expr]返回使用一个等式重写一个表达式的所有可能结果的列表，该函数将在另一个模块定义。一个表达式可能被重写为表达式本身（见习题 H），但是这样的可能导致无穷计算的重写是不允许的。函数 rws :: Expr -> [Step]返回以所有可能方式使用定律得到新表达式的单步计算列表，该列表通过轮流使用定律生成所有可能结果得到。这表示在计算中，定律的应用优先于子表达式，由此也解决了 12.1 节提出的担心的问题。只有实验能证明我们的选择是否正确。

312

函数 manyStep 使用 rws 构建尽可能多的步骤：

```
manyStep :: (Expr -> [Step]) -> Expr -> [Step]
manyStep rws e
  = if null steps then []
    else step : manyStep rws (snd step)
    where steps = rws e
          step  = head steps
```

当 rsw e 是空列表时，计算结束；否则，列表的第一个元素用于继续计算。

计算模块的其他函数处理显示和粘贴计算。一个计算的显示如下定义：

```
instance Show Calculation where
  showsPrec _ (Calc e steps)
    = showString "\n  " .
      shows e .
      showChar '\n' .
      compose (map showStep steps)
```

每个步骤显示如下：

```
showStep :: Step -> ShowS
showStep (why,e)
 = showString "=   {" .
   showString why .
   showString "}\n  " .
   shows e .
   showChar '\n'
```

为了把两个计算粘贴在一起，必须将一个计算的步骤取逆。例如，计算：

```
Calc e0 [(why1,e1),(why2,e2),(why3,e3)]
```

313

被转换成

```
Calc e3 [(why3,e2),(why2,e1),(why1,e0)]
```

特别是，一个计算的结论是取逆后的第一个表达式。下面表示如何将一个计算取逆：

```
reverseCalc :: Calculation -> Calculation
reverseCalc (Calc e steps)
 = foldl shunt (Calc e []) steps
   where shunt (Calc e1 steps) (why,e2)
         = Calc e2 ((why,e1):steps)
```

为了粘贴两个计算，首先检查两个计算的结论是否相同。如果不相同，则继续粘贴两个计算，并附带一个失败的标示：

```
  conc1
=  {... ??? ...}
  conc2
```

如果两个结论相同，还可以做得更聪明一点，而不是仅仅将两个计算粘在一起。如果一个计算的倒数第二个结论与另一个计算的倒数第二个结论匹配，那么可以将最后一步剪掉。下面是如何粘贴两个计算：

```
paste :: Calculation -> Calculation -> Calculation
paste calc1@(Calc e1 steps1) calc2
  = if conc1 == conc2
    then Calc e1 (prune conc1 rsteps1 rsteps2)
    else Calc e1 (steps1 ++ (gap,conc2):rsteps2)
```

```
where Calc conc1 rsteps1 = reverseCalc calc1
      Calc conc2 rsteps2 = reverseCalc calc2
      gap = "... ??? ..."
```

函数 prune 定义如下：

```
prune :: Expr -> [Step] -> [Step] -> [Step]
prune e ((_,e1):steps1) ((_,e2):steps2)
 | e1==e2 = prune e1 steps1 steps2
prune e steps1 steps2 = rsteps ++ steps2
 where Calc _ rsteps = reverseCalc (Calc e steps1)
```

最后是模块 Calculations 的声明：

```
module Calculations
  (Calculation (Calc), Step, calculate, paste)
```

314

```
where
import Expressions
import Laws
import Rewrites
import Utilities (compose)
```

模块输出了在主模块中定义 simplify 和 prove 需要的那些类型和函数。

12.5 重写

模块 Rewrites 的唯一目的是提供函数 rewrites 的定义，该函数出现在 calculate 定义中。回顾表达式 rewrites eqn e，它返回所有这些表达式的列表：将 e 中与 eqn 的左表达式匹配的某个子表达式用 eqn 的右表达式的适当特例替换。

有趣的是找出 rewrites 的定义。假如构造了一个表达式的所有子表达式的列表，可以用给定等式与列表的每个元素匹配，计算出构成匹配的替换（可能没有，也可能有一个或者多个，见 12.6 节关于匹配的内容），然后计算出代换后的新表达式。但是，如何在原表达式中用一个新的表达式代换一个子表达式？简单的回答是，没有办法，至少在没有确定每个子表达式在原表达式中的上下文或者位置之前是不可行的。若能确定子表达式的位置，则可将新子表达式插入这个位置。

方法不是显式地引入上下文，而是采取另一种途径。想法是钻进一个表达式中，在某个时刻对某个表达式应用重写，然后在爬出洞时构建重写的表达式。将需要一个工具函数 anyOne，其输入是能够生成一系列选择的函数，以及一个列表，然后为列表中一个元素设置一种选择。其定义如下：

```
anyOne :: (a -> [a]) -> [a] -> [[a]]
anyOne f []     = []
anyOne f (x:xs) = [x':xs | x' <- f x] ++
                  [x:xs' | xs' <- anyOne f xs]
```

或者为列表的第一个元素设置了一个选择，或者为第二个元素设置了一个选择，但不会为两个元素同时设置选择。例如，如果 f 1 =[-1, -2]和 f 2 =[-3, -4]，则有

315

```
anyOne f [1,2] = [[-1,2],[-2,2],[1,-3],[1,-4]]
```

以下是 rewrites 的定义：

```
rewrites :: Equation -> Expr -> [Expr]
rewrites eqn (Compose as) = map Compose (
   rewritesSeg eqn as ++ anyOne (rewritesA eqn) as)
rewritesA eqn (Var v) = []
rewritesA eqn (Con k es)
   = map (Con k) (anyOne (rewrites eqn) es)
```

定义的第一行表示，将当前表达式一段（segment）的重写与任意适当子表达式的重写串联。只有有参数的常数有子表达式。注意，anyOne 的两个应用具有不同的类型，一个应用于原子列表，一个应用于表达式列表。

接下来定义 rewritesSeg：

```
rewritesSeg :: Equation -> [Atom] -> [[Atom]]
rewritesSeg (e1,e2) as
   = [as1 ++ deCompose (apply sub e2) ++ as3
     | (as1,as2,as3) <- segments as,
       sub <- match (e1,Compose as2)]
```

函数 segmants 将列表化分成段：

```
segments as = [(as1,as2,as3)
              | (as1,bs)  <- splits as,
                (as2,as3) <- splits bs]
```

工具函数 splits 用各种可能方法拆分一个列表：

```
splits :: [a] -> [([a],[a])]
splits []     = [([],[])]
splits (a:as) = [([],a:as)] ++
                [(a:as1,as2) | (as1,as2) <- splits as]
```

例如：

```
ghci> splits "abc"
[("","abc"),("a","bc"),("ab","c"),("abc","")]
```

剩下的函数 apply 和 match 具有下列类型：

```
apply :: Subst -> Expr -> Expr
match :: (Expr,Expr) -> [Subst]
```

316

这两个函数分别定义在它们的模块 Substitutions 和 Matchings 中。最后是 Rewrites 的模块声明：

```
module Rewrites (rewrites)
where
import Expressions
import Laws (Equation)
import Matchings (match)
import Substitutions (apply)
import Utilities (anyOne, segments)
```

12.6 匹配

模块 Matching 的唯一目的是定义函数 match。该函数输入两个表达式，返回一个代换列表，在这些代换下第一个表达式可以转换为第二个表达式。如果两个表达式不匹配，则匹配不生成代换，但是，如果匹配成功，则可能生成多个代换。考虑匹配表达式

foo (f . g)与foo (a . b . c)。将f绑定下列4个表达式中任何一个，将g绑定相应的另一个，都可以使以上两个表达式匹配成功：

```
id,  a,  a . b,  a . b . c
```

尽管计算器每一步选择一个代换，但是在得到合法匹配过程中考虑多种代换很重要。例如，在匹配foo (f . g) . bar g和foo (a . b . c) . bar c时，子表达式f . g与a . b . c匹配，给出4种可能的代换。只有当bar g与bar c匹配时，其中的3个代换被拒绝。过早对第一次匹配的单个代换的承诺有可能导致错失一个成功匹配。

最直接定义match (e1,e2)的方法是将e1原子和e2原子的划分并排排列，第一个原子对应第一个划分段，第二个原子对应第二个划分段，等等。函数alignments具有下列类型：

```
alignments :: (Expr,Expr) -> [[(Atom,Expr)]]
```

317 并完成这种排列对应。定义这个函数需要将一个列表划分为给定段数的函数parts：

```
parts :: Int -> [a] -> [[[a]]]
parts 0 [] = [[]]
parts 0 as = []
parts n as = [bs:bss
             | (bs,cs) <- splits as,
               bss <- parts (n-1) cs]
```

有趣的是前两个子句：将空列表划分为0段有一种划分，即空划分，但是将非空列表划分为0段的划分不存在。例如：

```
ghci> parts 3 "ab"
[["","","ab"],["","a","b"],["","ab",""],
 ["a","","b"],["a","b",""],["ab","",""]]
```

现在可以定义：

```
alignments (Compose as,Compose bs)
 = [zip as (map Compose bss) | bss <- parts n bs]
   where n = length as
```

将每个原子和一个子表达式配对后，可以定义原子与表达式的匹配matchA：

```
matchA :: (Atom,Expr) -> [Subst]
matchA (Var v,e) = [unitSub v e]
matchA (Con k1 es1,Compose [Con k2 es2])
   | k1==k2 = combine (map match (zip es1 es2))
matchA _ = []
```

匹配变量永远成功，并给出单个代换。匹配两个常量时，只有常量名相同时匹配才成功。对于其他情况，matchA返回代换空列表。函数matchA依赖于match，该函数现在可以定义如下：

```
match :: (Expr,Expr) -> [Subst]
match = concatMap (combine . map matchA) . alignments
```

最后的配件是函数combine :: [[Subst]] -> [Subst]。combine的参数中每个代换列表代表可能的选择，所以combine必须用所有可能的方法在每个代换列表中选择一个代换，然后进行归一运算得到结果。下面将在代换模块中讨论这个函数。这样便完

成了 matches 的定义。模块的声明如下： 318

```
module Matchings (match)
where
import Expressions
import Substitutions (Subst, unitSub, combine)
import Utilities (parts)
```

函数 parts 被放在工具模块中，因为它不是特定于表达式的。

12.7 代换

一个代换是变量到表达式的有穷映射。关联列表是代换的一个简单表示：

```
type Subst = [(VarName,Expr)]
```

空代换和单个代换定义为：

```
emptySub    = []
unitSub v e = [(v,e)]
```

将一个代换应用于一个表达式可以得到另一个表达式：

```
apply :: Subst -> Expr -> Expr
apply sub (Compose as)
      = Compose (concatMap (applyA sub) as)
applyA sub (Var v)     = deCompose (binding sub v)
applyA sub (Con k es) = [Con k (map (apply sub) es)]
```

函数 binding 在一个非空代换中查找一个变量的绑定值：

```
binding :: Subst -> VarName -> Expr
binding sub v = fromJust (lookup v sub)
```

函数 lookup 在 Haskell 引导库 Prelude 中定义，如果没有绑定值，则返回 Nothing，如果 v 的绑定值是 e，则返回 Just e。函数 fromJust 在库 Data.Maybe 中定义，其功能是去掉包装 Just。

接下来解决 combine。该函数必须用所有可能的方法组合选择代换：在每个组成列表中选择一个代换，然后对这样的代换列表进行合一：

```
combine = concatMap unifyAll . cp
```

319

工具函数 cp 已经见过多次，它计算一个列表的列表的笛卡儿积。

函数 unifyAll 将一个代换列表合一。为定义这个函数，首先看如何将两个代换合一。如果两个代换相容，则合一结果是两个代换的并；如果两个代换不相容，则合一结果失败。为了处理失败的情况，可以使用 Maybe 类型，或者简单地用空列表或单元素列表。选择后者，只是因为 12.8 节将给出计算器的另一个定义，最简单的方法是统一使用基于列表的函数：

```
unify :: Subst -> Subst -> [Subst]
unify sub1 sub2 = if compatible sub1 sub2
                  then [union sub1 sub2]
                  else []
```

为定义 compatible 和 union，假定代换按照变量名的字典序表示。如果两个代换用同一个变量名与不同的表达式关联，则它们是不相容的：

```
compatible [] sub2 = True
compatible sub1 [] = True
compatible sub1@((v1,e1):sub1') sub2@((v2,e2):sub2')
   | v1<v2  = compatible sub1' sub2
   | v1==v2 = if e1==e2 then compatible sub1' sub2'
              else False
   | v1>v2  = compatible sub1 sub2'
```

并运算用同样的方式定义：

```
union [] sub2 = sub2
union sub1 [] = sub1
union sub1@((v1,e1):sub1') sub2@((v2,e2):sub2')
  | v1<v2  = (v1,e1):union sub1' sub2
  | v1==v2 = (v1,e1):union sub1' sub2'
  | v1>v2  = (v2,e2):union sub1  sub2'
```

函数 unifyAll 返回一个空列表或者单元素列表：

```
unifyAll :: [Subst] -> [Subst]
unifyAll = foldr f [emptySub]
   where f sub subs = concatMap (unify sub) subs
```

320 现在已经完成了所需要的所有定义。下面是模块声明：

```
module Substitutions
  (Subst, unitSub, combine, apply)
where
import Expressions
import Utilities (cp)
import Data.Maybe (fromJust)
```

这样，计算器总共由 9 个模块构成。

12.8 测试计算器

计算器的实用性如何呢？回答这个问题的唯一方法是在一些例子上试用计算器。这里将展示两个计算例子。第一个是第 5 章做过的关于数独中选择矩阵裁剪的计算。实际上是想证明：

```
filter (all nodups . boxs) . expand . pruneBy boxs
       = filter (all nodups . boxs) . expand
```

可以使用的定律如下：

```
defn pruneBy:       pruneBy f = f . map pruneRow . f
expand after boxs:  expand . boxs = map boxs . expand
filter with boxs:   filter (p . boxs)
                      = map boxs . filter p . map boxs
boxs involution:    boxs . boxs = id
map functor:        map f . map g = map (f.g)
map functor:        map id = id
defn expand:        expand = cp . map cp
filter after cp:    filter (all p) . cp = cp . map (filter p)
law of pruneRow:    filter nodups . cp . pruneRow
                      = filter nodups . cp
```

下面的计算完全是由计算器完成的，只是为了显示的方便，把某些表达式写成两个，这个任务应该由精美打印工具完成。无需研究细节，只要注意接近结尾的计算：

```
  filter (all nodups . boxs) . expand . pruneBy boxs
=   {filter with boxs}
```

```
  map boxs . filter (all nodups) . map boxs . expand .
  pruneBy boxs
=   {defn pruneBy}
  map boxs . filter (all nodups) . map boxs . expand .
  boxs . map pruneRow . boxs
=   {expand after boxs}
  map boxs . filter (all nodups) . map boxs . map boxs .
  expand . map pruneRow . boxs
=   {map functor}
  map boxs . filter (all nodups) . map (boxs . boxs) . expand .
  map pruneRow . boxs
=   {boxs involution}
  map boxs . filter (all nodups) . map id . expand .
  map pruneRow . boxs
=   {map functor}
  map boxs . filter (all nodups) . expand . map pruneRow . boxs
=   {defn expand}
  map boxs . filter (all nodups) . cp . map cp . map pruneRow . boxs
=   {map functor}
  map boxs . filter (all nodups) . cp . map (cp . pruneRow) . boxs
=   {filter after cp}
  map boxs . cp . map (filter nodups) . map (cp . pruneRow) . boxs
=   {map functor}
  map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
=   {law of pruneRow}
  map boxs . cp . map (filter nodups . cp) . boxs
=   {... ??? ...}
  map boxs . filter (all nodups) . map boxs . cp . map cp
=   {defn expand}
  map boxs . filter (all nodups) . map boxs . expand
=   {filter with boxs}
  filter (all nodups . boxs) . expand
```

321

是的，计算失败了。原因不难看出，需要在两个方向使用下列定律：

```
expand after boxs:    expand . boxs = map boxs . expand
```

但是，计算器做不到。

解决方法是进行人工分析，添加一个额外的定律：

```
hack:    map boxs . cp . map cp = cp . map cp . boxs
```

这个定律正是左右调换的 `expand after boxes` 定律，其中 `expand` 替换成了定义。这样一来计算器非常满意，生成了预期的结论：

```
....
  map boxs . cp . map (filter nodups . cp) . boxs
=   {map functor}
  map boxs . cp . map (filter nodups) . map cp . boxs
=   {filter after cp}
  map boxs . filter (all nodups) . cp . map cp . boxs
=   {hack}
  map boxs . filter (all nodups) . map boxs . cp . map cp
=   {defn expand}
  map boxs . filter (all nodups) . map boxs . expand
=   {filter with boxs}
  filter (all nodups . boxs) . expand
```

322

两种情况的计算都在几分之一秒内完成，所以效率似乎不是问题。除了那个人工分析以外，计算符合要求，几乎可以达到一个好的人工计算器的能力。

改进计算器

第二个例子更是雄心勃勃：使用计算器导出计算器的另一个版本。再次细看 `match` 的定义，定义依赖于 `combine`，后者涉及复杂的两个代换的合一，附带配套的相容性测试和并的计算。一个更好的想法是，只有当两个代换中有一个是单位代换时才计算两个代

换的并。然后一切变得更简单，或者更快。什么技术可以描述这种优化呢？是的，这是另一个累积参数的例子。就像累积参数可以避免使用昂贵的运算 ++ 一样，在此希望避免使用昂贵的运算 unify。

首先，下面是 match 的定义，使用了几个新的辅助函数：

```
match = concatMap matchesA . alignments
matchesA = combine . map matchA
matchA (Var v,e) = [unitSub v e]
matchA (Con k1 es1,Compose [Con k2 es2])
   | k1==k2 = matches (zip es1 es2)
matchA _ = []
matches = combine . map match
```

注意，这些函数的依赖关系呈环形：

```
match --> matchesA --> matchA --> matches --> match
```

这 4 个函数推广如下：

```
xmatch sub     = concatMap (unify sub) . match
xmatchA sub    = concatMap (unify sub) . matchA
xmatches sub   = concatMap (unify sub) . matches
xmatchesA sub = concatMap (unify sub) . matchesA
```

323 每个函数中新加的参数是一个累积参数。目的是找到这些函数的新定义，其环状依赖关系同前。

对于第一个计算，希望用 xmatch 重写 match，由此建立两组定义间的联系。为节省笔墨，将 concatMap 简记作 cmap。需要的 3 个定律是

```
defn xmatch:     xmatch s = cmap (unify s) . match
unify of empty:  unify emptySub = one
cmap of one:     cmap one = id
```

在第一个定律中必须写 s 而不写 sub（为什么?）；第二个定律是下列事实的点自由版本：

```
unify emptySub sub = [sub]
cmap one xs = concat [[x] | x <- xs] = xs
```

计算器不费力气便给出：

```
  xmatch emptySub
=  {defn xmatch}
  cmap (unify emptySub) . match
=  {unify of empty}
  cmap one . match
=  {cmap of one}
  match
```

下面再来处理 xmatchA。由于 matchA 定义中笨拙的模式匹配方式，因此仅仅给出一个简单的（人工）计算结果：

```
xmatchA sub (Var v,e) = concat [unify sub (unitSub v e)]
xmatchA sub (Con k1 es1,Compose [Con k2 es2])
    | k1==k2 = xmatches sub (zip es1 es2)
xmatchA _ = []
```

如果引入：

```
extend sub v e = concat [unify sub (unitSub v e)]
```

则容易导出：

```
extend sub v e
 = case lookup v sub of
     Nothing -> [(v,e):sub]
     Just e' -> if e==e' then [sub]
                else []
```

324

这里没有复杂的相容性测试，也没有两个代换的通用并。如上所承诺的，只有代换与单位代换的合一。

处理好 xmatchA 后，下面集中处理 4 个函数中的其他 3 个。就像用 xmatches 定义 xmatchA 一样，xmatch 可以用 xmatchesA 定义。特别地，想证明：

```
xmatch s = cmap (xmatchesA s) . alignments
```

下面是需要的定律：

```
defn match:      match = cmap matchesA . alignments
defn xmatch:     xmatch s = cmap (unify s) . match
defn xmatchesA:  xmatchesA s = cmap (unify s) . matchesA
cmap after cmap: cmap f . cmap g = cmap (cmap f . g)
```

最后一个纯粹的组合定律是新的，定律的验证留作练习。计算器生成下列计算：

```
  xmatch s
=   {defn xmatch}
  cmap (unify s) . match
=   {defn match}
  cmap (unify s) . cmap matchesA . alignments
=   {cmap after cmap}
  cmap (cmap (unify s) . matchesA) . alignments
=   {defn xmatchesA}
  cmap (xmatchesA s) . alignments
```

到目前为止，一切顺利。现在4 个函数中还有两个没有定义，即 xmatches 和 xmatchesA。我们希望每个函数得到一个递归定义，而且不涉及 unify。这两个函数的定义及其相像，很有可能任何一个函数的计算可立即用于另一个函数。当然，这种元计算思想是这种计算器不可及的。

现在集中注意力于 xmatchesA。首先将 xmatchesA 转换为点自由式，删除上面定义中的参数 s。修改后的定义是：

```
xmatchesA :: (Subst,[(Atom,Expr)]) -> Subst
xmatchesA = cup . (one * matchesA)
cup = cmap unify . cpp
```

其中组合函子定义为：

```
cpp (xs,ys) = [(x,y) | x <- xs, y <- ys]
```

因此，有

325

```
  xmatchesA (sub,aes)
= cup ([sub],aes)
= concat [unify (s,ae) | s <- [sub],ae <- matchesA aes]
= concat [unify (sub,ae) | ae <- matchesA aes]
```

除了假定 unify 现在是非卡瑞函数外，这是 xmatchesA 定义的忠实点自由式翻译。

新的函数 cup 具有类型[Subst] -> [Sbust] -> [Subst]。后面将说明 cup 满足结合律，这是 unify 永远不会具有的性质（为什么不会?）。如在第 7 章所述，累积参

数技术需要所关心的运算具有结合性。

首先需要检查对于新的定义，前面的计算仍然是正确的。假定给出下面定律：

```
defn match:      match = cmap matchesA . alignments
defn xmatch:     xmatch = cup . (one * match)
defn xmatchesA:  xmatchesA = cup . (one * matchesA)
```

计算器便可生成

```
  xmatch
=   {defn xmatch}
  cup . (one * match)
=   {defn match}
  cup . (one * (cmap matchesA . alignments))
=   {... ??? ...}
  cmap (cup . (one * matchesA)) . cpp . (one * alignments)
=   {defn xmatchesA}
  cmap xmatchesA . cpp . (one * alignments)
```

啊，计算通不过。细看计算中的缺口，似乎既需要* 的双函子律，也需要关于 cmap 和 cup 的下面断言：

```
cross bifunctor: (f * g) . (h * k) = (f . h) * (g . k)
cmap-cup: cmap (cup . (one * g)) . cpp = cup . (id * cmap g)
```

现在计算器很满意，给出下列计算：

```
  xmatch
=   {defn xmatch}
  cup . (one * match)
=   {defn match}
  cup . (one * (cmap matchesA . alignments))
=   {cross bifunctor}
  cup . (id * cmap matchesA) . (one * alignments)
=   {cmap-cup}
  cmap (cup . (one * matchesA)) . cpp . (one * alignments)
=   {defn xmatchesA}
  cmap xmatchesA . cpp . (one * alignments)
```

326

接下来需要考虑刚才的断言，除了它让计算器给出期望的结果外，没有其他理由支持其正确性。但是，可以使用另一个并非特定于匹配的定律，让计算器证明断言。这个证明留作习题 M。定义另外两个定律：

```
defn cup:     cup = cmap unify . cpp
cmap-cpp: cmap (cpp . (one * f)) . cpp = cpp . (id * cmap f)
```

然后计算器生成下列计算：

```
  cmap (cup . (one * g)) . cpp
=   {defn cup}
  cmap (cmap unify . cpp . (one * g)) . cpp
=   {cmap after cmap}
  cmap unify . cmap (cpp . (one * g)) . cpp
=   {cmap-cpp}
  cmap unify . cpp . (id * cmap g)
=   {defn cup}
  cup . (id * cmap g)
```

现在可以了。看起来 cmap-cup 定律是正确的，而且今后还会用到这个定律。现在回到主要任务，即将 xmatchesA 递归地用下列形式的两个等式表示：

```
xmatchesA . (id * nil)  = ...
xmatchesA . (id * cons) = ...
```

希望这种定义不涉及 unify。

为了达到这个目的，还不清楚需要什么样的定律。下面将写出想到的可能有用的定律。第一组由 4 个主要定义构成：

```
defn match:      match = cmap matchesA . alignments
defn matchesA:   matchesA = combine . map matchA
defn xmatch:     xmatch    = cup . (one * match)
defn xmatchesA:  xmatchesA = cup . (one * matchesA)
defn xmatchA:    xmatchA   = cup . (one * matchA)
defn combine:    combine = cmap unifyAll . cp
```

第二组是关于 cmap 的一些新定律：

```
cmap after map:    cmap f . map g = cmap (f . g)
cmap after concat: cmap f . concat = cmap (cmap f)
cmap after nil:    cmap f . nil = nil
cmap after one:    cmap f . one = f
```

第三组是关于 map 的一些新定律： 327

```
map after nil: map f . nil = nil
map after one: map f . one = one . f
map after cons: map f . cons = cons . (f * map f)
map after concat: map f . concat = concat . map (map f)
```

第四组是关于 cup 的定律：

```
cup assoc:  cup . (id * cup) = cup . (cup * id) . assocl
cup ident:  cup . (f * (one . nil)) = f . fst
cup ident:  cup . ((one . nil) * g) = g . snd
assocl: assocl. (f * (g * h)) = ((f * g) * h) . assocl
```

最后，加上下列定义和定律：

```
cross bifunctor: (f * g) . (h * k) = (f . h) * (g . k)
cross bifunctor: (id * id) = id
defn cp:  cp . nil = one . nil
defn cp:  cp . cons = map cons . cpp . (id * cp)
defn unifyAll: unifyAll . nil = one . nil
defn unifyAll: unifyAll . cons = cup . (one * unifyAll)
unify after nil:  unify . (id * nil) = one . fst
```

以上总共列出了 30 个定律（包括没有重复列出的定律：map 的 2 个函子律和 cmap 的 3 个定律）。下面祈祷并希望：

```
  xmatchesA . (id * nil)
=   {defn xmatchesA}
  cup . (one * matchesA) . (id * nil)
=   {cross bifunctor}
  cup . (one * (matchesA . nil))
=   {defn matchesA}
  cup . (one * (combine . map matchA . nil))
=   {map after nil}
  cup . (one * (combine . nil))
=   {defn combine}
  cup . (one * (cmap unifyAll . cp . nil))
=   {defn cp}
  cup . (one * (cmap unifyAll . one . nil))
=   {cmap after one}
  cup . (one * (unifyAll . nil))
=   {defn unifyAll}
  cup . (one * (one . nil))
=   {cup ident}
  one . fst
```

结果令人满意。前面已经证明了 xmatchesA sub [] = [sub]。但是，递归情况不容易建立起来。为此，必须猜想结果，并设法证明。下面是期望的结果，首先用点式表示，然后用点自由式表示：

328
```
xmatchesA sub (ae:aes)
  = concat [xmatchesA sub' aes | sub' <- xmatchA sub ae]

xmatchesA . (id * cons)
 = cmap xmatchesA . cpp . (xmatchA * one) . assocl
```

可以对右式进行简化（暂时从 laws2 中去掉 xmatchA 和 matchesA 的定义）：

```
  cmap xmatchesA . cpp . (xmatchA * one) . assocl
=   {defn xmatchesA}
  cmap (cup . (one * matchesA)) . cpp . (xmatchA * one) . assocl
=   {cmap-cup}
  cup . (id * cmap matchesA) . (xmatchA * one) . assocl
=   {cross bifunctor}
  cup . (xmatchA * (cmap matchesA . one)) . assocl
=   {cmap after one}
  cup . (xmatchA * matchesA) . assocl
```

现在希望证明：

```
xmatchesA . (id * cons)
     = cup . (xmatchA * matchesA) . assocl
```

但是，不幸的是计算器做不到这点。缺口在这里：

```
  cup . ((cup . (one * matchA)) * matchesA)
=  {... ??? ...}
  cup . (one * (cup . (matchA * matchesA))) . assocl
```

这个缺口很容易手动消除：

```
  cup . ((cup . (one * matchA)) * matchesA)
=  {cross bifunctor (backwards)}
  cup . (cup * id) . ((one * matchA) * matchesA)
=  {cup assoc}
  cup . (id * cup) . assocl . ((one * matchA) * matchesA)
=  {assocl}
  cup . (id * cup) . (one * (matchA * matchesA)) . assocl
=  {cross bifunctor}
  cup . (one * (cup . (matchA * matchesA))) . assocl
```

又一次看到，不能双向应用定律是问题的起因。在这里加上了注释“手工完成的作品!”，而没有设法强迫定律写成计算器可以接受的形式。

为了结束这个例子，下面给出计算出的程序：

```
match = xmatch emptySub
xmatch sub (e1,e2)
```
329
```
 = concat [xmatchesA sub aes | aes <- alignments (e1,e2)]
xmatchesA sub [] = [sub]
xmatchesA sub (ae:aes)
 = concat [xmatchesA sub' aes | sub' <- xmatchA sub ae]

xmatchA sub (Var v,e) = extend sub v e
xmatchA sub (Con k1 es1,Compose [Con k2 es2])
    | k1==k2 = xmatches sub (zip es1 es2)
xmatchA _ = []
```

缺席的定义是 xmatches。不过，处理 xmatchesA 的过程完全适用于处理 matches，最后得到

```
xmatches sub [] = [sub]
xmatches sub ((e1,e2):es)
 = concat [xmatches sub' es | sub' <- xmatch sub (e1,e2)]
```

结论

这两个练习的正面结论是，计算器确实可以用来辅助构建形式证明。但是，在这个过程中需要大量的人工输入，设置合适的定律，识别辅助断言和控制计算进行的次序。主要的负面结论是，计算器不能双向应用定律是一个大的缺陷。函子律是问题的主要来源，但是其他定律也有类似问题（参见习题中的例子）。计算器可以从几个方面改进，但是进一步改进讨论将放在习题中。

关于这个计算器还有 3 个方面值得提出。第一，完整的计算器只有 450 行 Haskell 代码，改进的版本更短。仅这一点已经证明了函数式程序设计的表达能力。第二，将定律表达为纯函数等式，然后使用等式逻辑做证明似乎是切实可行的方法。为此，必须花些时间将定义表达成点自由形式，但是，一旦这个工作完成，等式逻辑计算变得非常高效。

第三，除语法分析外，单子没有出现在计算器中。实际上，计算器较早的版本确实使用了单子，但是，它们渐渐被淘汰了。一个原因是，我们发现没有单子的代码更简单，效率也没有很大的损失；另一个原因是我们想为习题中计算器的改进留出空间。单子对于涉及与外界交互的许多应用是绝对必要的，但是在纯函数方式已经很有效时，单子的使用可能变得多余。 330

用以上几点结论结束本章。

12.9 习题

习题 A 假如我们想让 `calculate` 返回所有可能计算构成的树。使用什么样的树合适？

习题 B 为什么下列定律永远不被使用，至少如果定律用下列形式给出的话？

```
map (f . g)  = map f . map g
cmap (f . g) = cmap f . map g
```

习题 C 以下是计算器给出的一个计算：

```
  map f . map g h
=   {map functor}
  map (f . g)
```

请解释这种奇怪、明显没意义的结果。对计算器做怎样的简单改动可以避免这种不合理计算的问题？

习题 D 与习题 C 类似的问题，对于计算器的严肃批评指出，错误信息是完全不透明的。例如：

```
parse law "map f . map g = map (f . g)"
parse law "map functor: map f . map g   map (f . g)"
```

这两个计算都引发神秘的错误信息。请问是什么错误信息？在一个计算中使用下列定律的效果是什么？

```
strange: map f . map g = map h
```

331

同样，如何修改计算器使其避免接受这种定律？

习题 E 原子的 `showsPrec` 定义使用了之前没有用到的 Haskell 的一个事实。同样的

机制后来在计算器的混合模式匹配与条件等式的函数中用到。这种机制是什么？

习题 F 定义：

```
e1 = foo (f . g) . g
e2 = bar f . baz g
```

请列出当 rewrites (e1,e2)应用于表达式 foo (a . b . c) . c 时所生成的表达式。请问计算器会选择哪一个？

习题 G 计算器能够成功地将 foo f . foo f 与下列表达式匹配吗？

```
foo (bar g h) . foo (bar (daz a) b) ?
```

习题 H 书中曾称这是可能的：应用一个完全合法的非平凡定律将使得某些表达式不变。请给出一个这样的定律和一个表达式，将定律应用于表达式重写成自身。

习题 I 书中 rewrites 定义中的函数 anyOne 设置单个选择，但是为什么不使用 everyOne同时设置每一个选择？因此，如果 f 1 = [-1, -2], f 2 = [-3, -4]，那么

```
everyOne f [1,2] = [[-1,-3],[-1,-4],[-2,-3],[-3,-4]]
```

使用 everyOne 而不是 anyOne 意味着每个重写将被应用于匹配一个定律的每个可
332 能的表达式。请给出 everyOne 的定义。

习题 J 长度为 n 的列表有多少个段？rewritesSeq 的定义是低效的，因为在长度为 n 的列表的分段中，空段作为中间分段出现 $n+1$ 次。这表示与 id 的匹配进行了 $n+1$ 次，而不是一次。如何重写 segments 来消除这些重复工作？

习题 K 证明 cmap f . cmap g = cmap (cmap f . g)。需要的定律如下：

```
defn cmap:         cmap f = concat . map f
map functor:       map f . map g = map (f.g)
map after concat: map f . concat = concat . map (map f)
concat twice:      concat . concat = concat . map concat
```

习题 L 定律 cmap-cpp 如下：

```
cmap (cpp . (one * f)) . cpp = cpp . (id * cmap f)
```

请利用下列定律证明 cmap-cpp。

```
cmap after cmap:  cmap f . map g = cmap (f . g)
cmap after cpp:  cmap cpp . cpp = cpp . (concat * concat)
cross bifunctor: (f * g) . (h * k) = (f . h) * (g . k)
map after cpp:   map (f * g) . cpp = cpp . (map f * map g)
defn cmap:       cmap f = concat . map f
concat after id: concat . map one = id
```

计算器能够完成这个证明吗？

12.10 答案

习题 A 答案 用表达式作为结点的标记，定律名作为边的标记。由此得到

```
type Calculation = Tree Expr LawName
data Tree a b    = Node a [(b,Tree a b)]
```
333

习题 B 答案 这样会使得计算器陷入无穷计算。例如：

```
  map foo
= {map functor}
  map foo . map id
= {map functor}
  map foo . map id . map id
```

等等。

习题 C 答案 根据语法规则，表达式 `map f . map g h` 是完全合法的，但是，当然不应该。计算器不会限制每个常量和同一个常量的每次出现都具有相同个数的参数。函子律可以成功匹配该表达式的原因是，在 `matchA` 的定义中，函数 `zip` 将第二个 `map` 的两个参数截短为一个。一个更好的计算器应该检查每个常量具有固定数目的参数。

习题 D 答案 神秘的信息是“空列表的第一个元素”。第一个分析失败，因为定律缺少名，第二个缺少等号。使用奇怪的定律将引发计算器产生错误信息，因为与左边模式匹配后 `h` 没有绑定任何表达式，当系统需要 `h` 的绑定值时引起错误。为了避免这种问题，计算器应该检查一个定律右边的每个变量都在左边出现。

习题 E 答案 `showsPrec` 的定义如下：

```
showsPrec p (Con f [e1,e2])
  | isOp f   = expression1 e1 e2
showsPrec p (Con f es)
  = expression2 es
```

一个更数学化的定义是

```
showsPrec p (Con f [e1,e2])
  | isOp f    = expression1 e1 e2
  | otherwise = expression2 [e1,e2]
showsPrec p (Con f es) = expression2 es
```

334

要点是，在一个子句中，如果模式与参数不匹配，或者模式与参数匹配，但是条件不成立，则本子句被禁止，继续选择后面的子句。

习题 F 答案 有两个重写，不是一个：

```
bar (a . b . c) . baz id . c
bar (a . b) . baz c
```

计算器将选择匹配的第一个子表达式，这也表示第一个重写被选中。或者更好的方法是让 `rewritesSeg` 先应用于长段，后应用于较短的段。

习题 G 答案 不能，用给出的 `match` 定义不行。只有当 `g` 绑定 `daz a`，`h` 绑定 `b` 时，将 `f` 绑定表达式 `bar (daz a) b` 可以匹配，但是给出的 `match` 定义不能进行完全合一。

习题 H 答案 有许多例子，这里只是一个例子。考虑下列定律：

```
if p f g . h = if (p . h) (f . h) (g . h)
```

左边匹配 `if a b c`，其中 `h` 绑定 `id`，而且结果还是同一个表达式。

习题 I 答案 可能的诱惑是定义：

```
everyOne f = cp . map f
```

但是，如果 `f` 不返回任何选择，则该定义不可行。必须定义：

```
everyOne :: (a -> [a]) -> [a] -> [[a]]
everyOne f   = cp . map (possibly f)
possibly f x = if null xs then [x] else xs
               where xs = f x
```

335 在这个定义中，f 返回选择的非空列表。

习题 J 答案 长度为 n 的列表有 $(n+1)(n+2)/2$ 个段。改进的定义如下：

```
segments xs = [([],[],xs] ++
              [(as,bs,cs)
              | (as,ys) <- splits xs,
                (bs,cs) <- tail (splits ys)]
```

习题 K 答案 计算器输出：

```
  cmap f . cmap g
=   {defn cmap}
  concat . map f . cmap g
=   {defn cmap}
  concat . map f . concat . map g
=   {map after concat}
  concat . concat . map (map f) . map g
=   {map functor}
  concat . concat . map (map f . g)
=   {concat after concat}
  concat . map concat . map (map f . g)
=   {map functor}
  concat . map (concat . map f . g)
=   {defn cmap}
  concat . map (cmap f . g)
=   {defn cmap}
  cmap (cmap f . g)
```

习题 L 答案 人工证明如下：

```
  cmap (cpp . (one * g)) . cpp
=   {cmap after cmap (backwards)}
  cmap cpp . map (one * g) . cpp
=   {map after cpp}
  cmap cpp . cpp . (map one * map g)
=   {cmap after cpp}
  cpp . (concat * concat) . (map one * map g)
=   {cross bifunctor}
  cpp . ((concat . map one) * concat (map g))
=   {defn cmap (backwards)}
  cpp . ((concat . map one) * cmap g)
=   {concat after id}
  cpp . (id * cmap g)
```

不能，这个计算不能自动完成。定律 cmap after cmap 反向写会引起计算器陷入循环（见习题 B）。

12.11 注记

336 ~ 337 本章的计算器基于我的牛津同事 Mike Spivey 未公开的一个定理证明器。伦敦城市大学的 Ross Paterson 设计了具有内置的可以双向应用的函子定律的版本。

一个最先进的证明辅助器是 Coq，参见 http://coq. inria. fr/。

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

B

C

D

E

F

G

H

I

J

K

L

O

P

Q

R

S